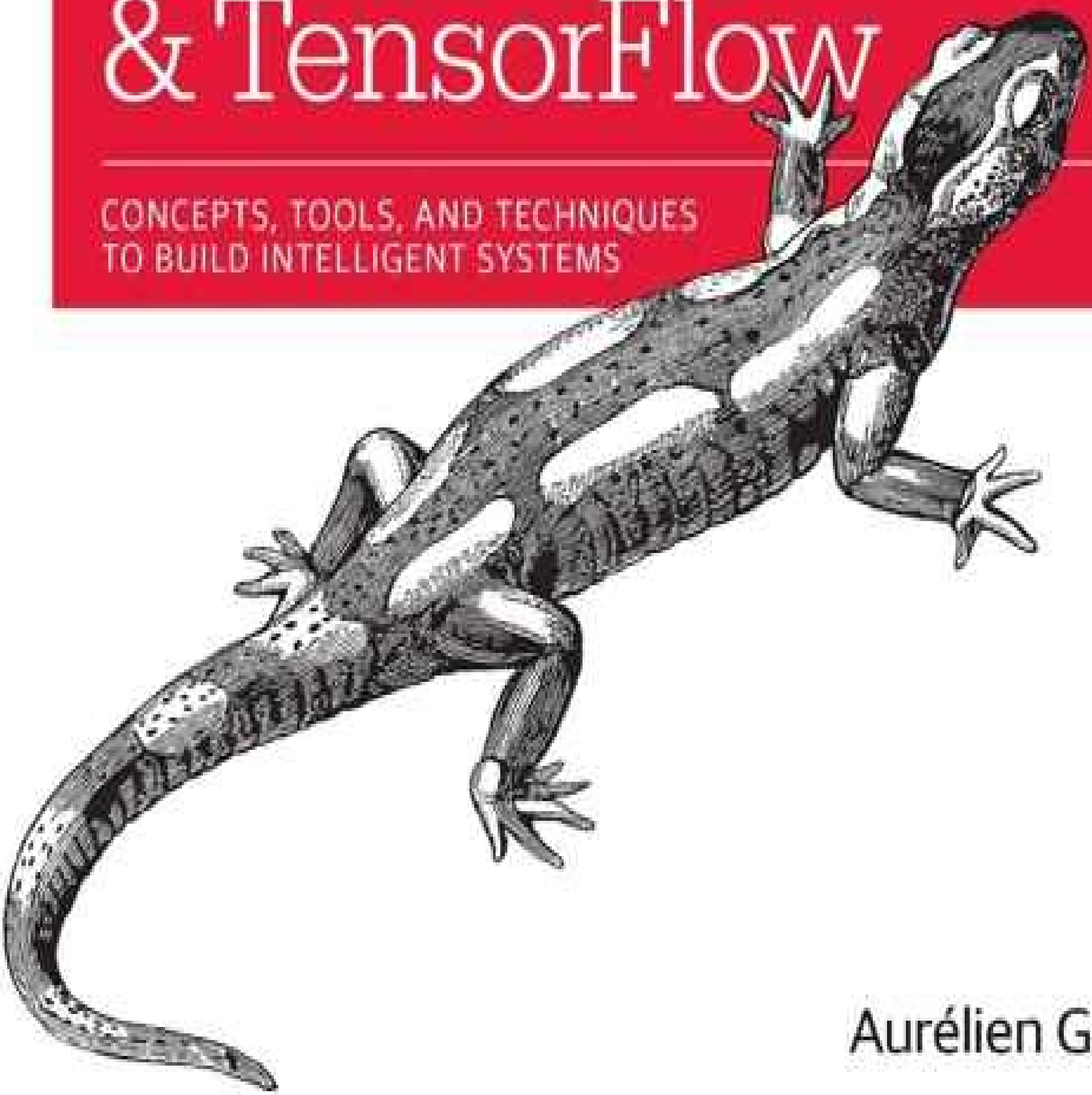
O'REILLY

Hands-On Machine Learning with Scikit-Learn & TensorFlow

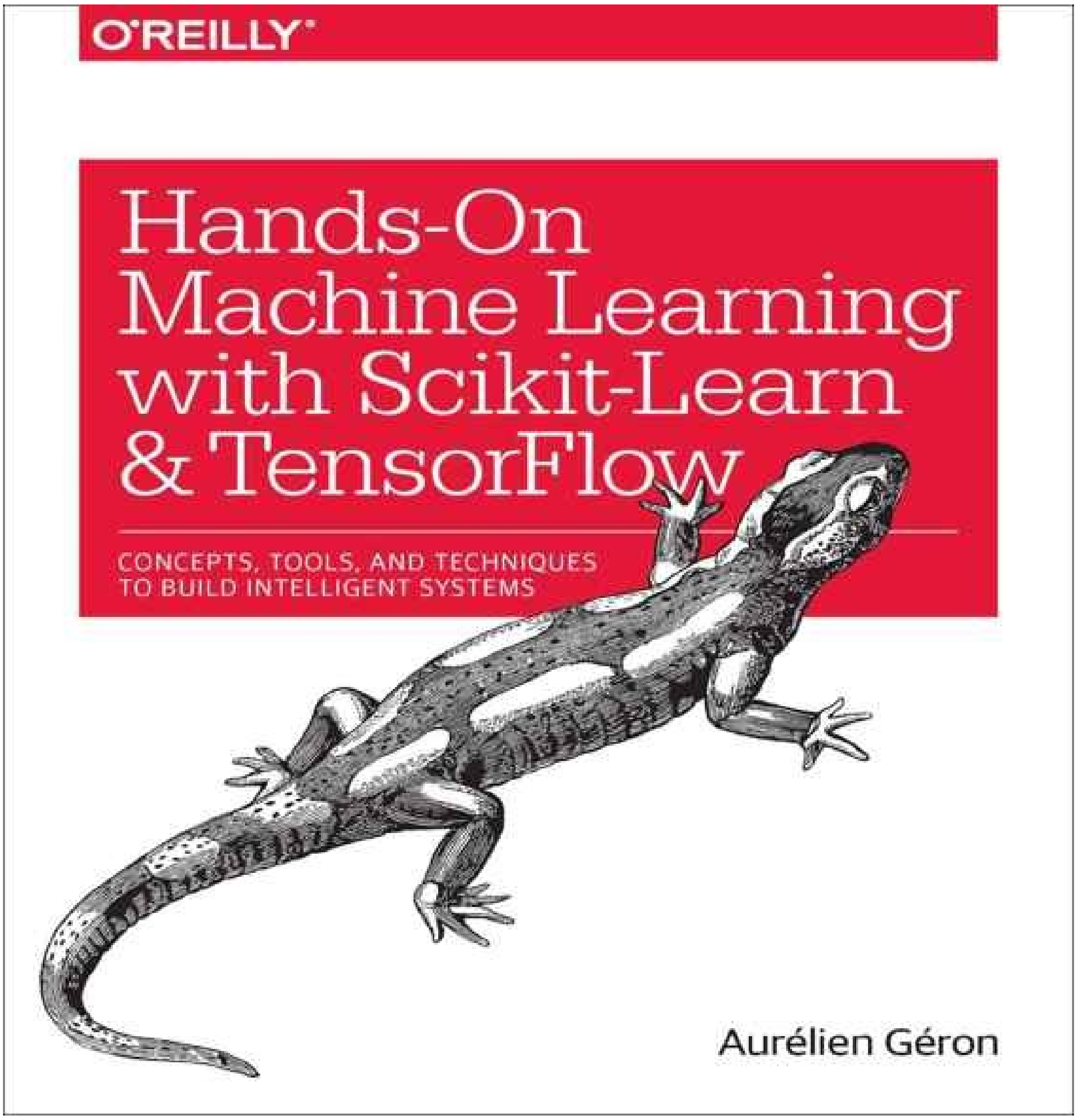
CONCEPTS, TOOLS, AND TECHNIQUES TO BUILD INTELLIGENT SYSTEMS



Aurélien Géron

目錄

Sklearn 与 TensorFlow 机器学习实用指南



> 欢迎任何人参与和完善：一个人可以走的很快，但是一群人却可以走的更远

- **ApacheCN - 学习机器学习群【629470233】**
- **Machine Learning in Action（机器学习实战）| ApacheCN（apache 中文网）**
- **@SeanCheney** 翻译的《利用 **Python** 进行数据分析 第二版》

- 在线阅读
- PDF格式
- EPUB格式
- MOBI格式
- 代码仓库

目录结构

第一部分 机器学习基础

第二部分 神经网络与深度学习

联系方式

> 项目负责人

- @SeanCheney: 731384963
- @飞龙: 562826179
- @小瑶: 190442212

> 项目贡献者

标题	译者	校对
前言	@小瑶	@小瑶
第一部分 机器学习基础	-	
一、机器学习概览	@SeanCheney	@Lisanaaa @飞龙 @yanmengk @Liu Shangfeng
二、一个完整的机器学习项目	@SeanCheney	@Lisanaaa @飞龙 @PeterHo
三、分类	@时间魔术师	@Lisanaaa @飞龙 @ZTFrom1994
四、训练模型	@C-PIG	@PeterHo @飞龙
五、支持向量机	@QiaoXie	@飞龙 @PeterHo @yanmengk
六、决策树	@Lisanaaa @y3534365	@飞龙
七、集成学习和随机森林	@friedhelm739	@飞龙 @PeterHo @yanmengk
八、降维	@loveSnowBest	@飞龙 @PeterHo @yanmengk
第二部分 神经网络与深度学习	-	
九、启动并运行TensorFlow	@akonwang @WilsonQu	@Lisanaaa @飞龙
十、人工神经网络介绍	@akonwang @friedhelm739	@飞龙
十一、训练深层神经网络	@akonwang @飞龙	@飞龙 @Zeyu Zhong
十二、设备和服务器上的分布式TensorFlow	@空白	@飞龙
十三、卷积神经网络	@akonwang @WilsonQu	@飞龙
十四、循环神经网络	@akonwang @alexcheen @飞龙	@飞龙
十五、自编码器	@akonwang	@飞龙 @yanmengk
十六、强化学习	@friedhelm739	@飞龙
其它	@片刻	

编译

```
gitbook install # 安装必要的插件
gitbook serve   # 编译 HTML
gitbook epub    # 编译 EPUB
```

免责声明

ApacheCN 纯粹出于学习目的与个人兴趣翻译本书，不追求任何经济利益。

本译文只供学习研究参考之用，不得用于商业用途。ApacheCN 保留对此版本译文的署名权及其它相关权利。

赞助我们



ApacheCN 组织资源

深度学习	机器学习	大数据	运维工具
TensorFlow R1.2 中文文档	机器学习实战-教学	Spark 2.2.0和2.0.2 中文文档	Zeppelin 0.7.2 中文文档
Pytorch 0.3 中文文档	Sklearn 0.19 中文文档	Storm 1.1.0和1.0.1 中文文档	Kibana 5.2 中文文档
	LightGBM 中文文档	Kudu 1.4.0 中文文档	
	XGBoost 中文文档	Elasticsearch 5.4 中文文档	
	kaggle: 机器学习竞赛	Beam 中文文档	

第0章 前言

1、机器学习海啸

2006年，Geoffrey Hinton等人发表了一篇论文，展示了如何训练能够识别具有最新精度（>98%）的手写数字的深度神经网络。他们称这种技术为“Deep Learning”。当时，深度神经网络的训练被广泛认为是不可能的，并且大多数研究人员自 20 世纪 90 年代以来就放弃了这个想法。这篇论文重新激起了科学界的兴趣，不久之后，许多新发表的论文表明，深度学习不仅是可能的，而且能够取得其他的 Machine Learning 技术都难以匹配的令人兴奋的成就（借助巨大的计算能力和大量的数据）。这种热情很快扩展到机器学习的许多的其他领域。

Deep Learning 快速发展的 10 年间和机器学习已经征服了这个行业：它现在成为了当今高科技产品中的许多黑科技的核心，比如，为您的网络搜索结果排名，为智能手机的语音识别提供支持，为您推荐您喜欢的视频，在 Go 游戏中击败世界冠军。在你知道之前，它都可能会驾驶您的汽车。

2、您项目中的机器学习

现在你是不是对机器学习感到兴奋，并且很乐意加入到这个阵营中？ 也许你希望给自己制造的机器人赋予一个自己的大脑？让它可以面部识别？还是学会到处走走？

也许你的公司有大量的数据（用户日志，财务数据，生产数据，机器传感器数据，热线统计数据，人力资源报告等），如果你知道在哪方面观察，你可能会发现一些隐藏着的瑰宝。例如：

- 细分客户，为每个团队找到最佳的营销策略
- 根据类似客户购买的产品为每个客户推荐产品
- 检测哪些交易可能是欺诈行为
- 预测下一年的收入
- 更多应用

无论什么原因，你决定开始学习机器学习，并在你的项目中实施，这是一个好主意！

3、目标和方法

本书假定你对机器学习几乎一无所知。它的目标是给你实际实现能够从数据中学习的程序所需的概念，直觉和工具。

我们将介绍大量的技术，从最简单的和最常用的（如线性回归）到一些定期赢得比赛的深度学习技术。

我们将使用现成的 Python 框架，而不是实现我们自己的每个算法的玩具版本：

- Scikit-learn 非常易于使用，并且实现了许多有效的机器学习算法，因此它为学习机器学习提供了一个很好的切入点。

- TensorFlow 是使用数据流图进行分布式数值计算的更复杂的库。它通过在潜在的数千个多 GPU 服务器上分布式计算，可以高效地训练和运行非常大的神经网络。TensorFlow 是被 Google 创造的，支持其大型机器学习应用程序。于 2015年11月开源。

本书倾向于实际操作的方法，通过具体的实例和一点理论来增加对机器学习的直观理解。虽然你可以在不拿笔记本电脑的情况下阅读此书，但是我们强烈建议你通过 https://github.com/ageron/handson-ml 在线实现 Jupyter notebooks 上的代码示例。

4、准备条件

本书假定您有一些 Python 编程经验，并且比较熟悉 Python 的主要科学库，特别是 NumPy，Pandas 和 Matplotlib 。

另外，如果你关心的是底层实现/原理，你应该对大学水平的数学（微积分，线性代数，概率和统计学）有一些了解。

如果你还不了解 Python，http://learnpython.org/ 是你学习使用 Python 的好地方。python.org 官方教程也是相当不错的。

如果你从未使用过 Jupyter ，第 2 章将指导你完成安装和基本操作：它是你工具箱中的一个很好的工具。

如果你不熟悉 Python 的科学库，提供的一些 Jupyter notebook 包括了一些教程。还有一个线性代数的快速数学教程。

5、路线图

这本书分为两个部分。

第一部分，机器学习的基础知识，涵盖以下主题：

- 什么是机器学习？它被试图用来解决什么问题？机器学习系统的主要类别和基本概念是什么？
- 典型的机器学习项目中的主要步骤。
- 通过拟合数据来学习模型。
- 优化成本函数（cost function）。

- 处理，清洗和准备数据。
- 选择和设计特征。
- 使用交叉验证选择一个模型并调整超参数。
- 机器学习的主要挑战，特别是欠拟合和过拟合（偏差和方差权衡）。
- 对训练数据进行降维以对抗 the curse of dimensionality（维度诅咒）
- 最常见的学习算法：线性和多项式回归， Logistic 回归，k-最近邻，支持向量机，决策树，随机森林和集成方法。

第二部分，神经网络和深度学习，包括以下主题：

- 什么是神经网络？它们有啥优势？
- 使用 TensorFlow 构建和训练神经网络。
- 最重要的神经网络架构：前馈神经网络，卷积网络，递归网络，长期短期记忆网络（LSTM）和自动编码器。
- 训练深度神经网络的技巧。
- 对于大数据集缩放神经网络。
- 强化学习。

第一部分主要基于 scikit-learn ，而第二部分则使用 TensorFlow 。

注意：不要太急于深入学习到核心知识：深度学习无疑是机器学习中最令人兴奋的领域之一，但是你应该首先掌握基础知识。而且，大多数问题可以用较简单的技术很好地解决（而不需要深度学习），比如随机森林和集成方法（我们会在第一部分进行讨论）。如果你拥有足够的数据，计算能力和耐心，深度学习是最适合复杂的问题的，如图像识别，语音识别或自然语言处理。

6、其他资源

有许多资源可用于了解机器学习。Andrew Ng 在 Coursera 上的 ML 课程和 Geoffrey Hinton 关于神经网络和深度学习的课程都是非常棒的，尽管这些课程需要大量的时间投入（大概是几个月）。

还有许多关于机器学习的比较有趣的网站，当然还包括 scikit-learn 出色的 用户指南。你可能会喜欢上 Dataquest ，它提供了一个非常好的交互式教程，还有 ML 博客，比如那些在 Quora 上列出来的博客。最后，Deep Learning 网站 有一个很好的资源列表来学习更多。

当然，还有很多关于机器学习的其他介绍性书籍，特别是：

- Joel Grus, Data Science from Scratch (O'Reilly). 这本书介绍了机器学习的基础知识，并在纯 Python 中实现了一些主要算法（从名字上看就可以知道，从头开始）。

- Stephen Marsland, Machine Learning: An Algorithmic Perspective (Chapman andHall). 这本书对机器学习有一个很好的介绍，涵盖了广泛的主题，Python 中的代码示例（也是从零开始，但是使用 NumPy）。

- Sebastian Raschka, Python Machine Learning (Packt Publishing). 本书也对机器学习有一个很好的介绍，但是利用了 Python 的开源库（Pylearn 2 和 Theano）。
- Yaser S. Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin, Learning fromData (AMLBook). 对 ML 有一个相对理论化的介绍，这本书提供了比较深刻的见解，特别是 bias/variance tradeoff （偏差/方差 权衡）（见第 4 章）。
- Stuart Russell and Peter Norvig, Artificial Intelligence: A Modern Approach, 3rd Edition (Pearson). 这是一本很好的（并且很大）的书，涵盖了包括机器学习在内的大量主题。这有助于更加深刻地理解 ML 。

最后，一个很好的学习方法就是加入 ML 竞赛网站，例如 kaggle.com ，这样可以让你在现实世界的问题上锻炼自己的技能，并从一些最好的 ML 专业人士那里获得帮助和见解。

7、本书中的一些约定

本书使用以下印刷约定：

- 斜体 —— 指示新术语，网址，电子邮件地址，文件名和文件扩展名。
- 等宽 —— 用于程序清单，以及段落内用于引用程序元素，如变量或函数名称，数据库，数据类型，环境变量，语句和关键字。
- 等宽粗体 —— 显示应由用户逐字输入的命令或其他文本。
- 等宽斜体 —— 显示应由用户提供的值或由上下文确定的值替换的文本。
- 小松鼠图标 —— 此元素表示一个小提示或建议。
- 小乌鸦图标 —— 此元素表示一个普通的说明。
- 小蝎子图标 —— 此元素表示一个警告和注意。

8、使用代码示例

补充材料（代码示例，练习题等）可以从 https://github.com/ageron/handson-ml 下载。

这本书是为了帮助你完成工作。一般来说，如果本书提供了示例代码，则可以在程序和文档中使用它。除非你复制了大部分代码，否则你无需联系我们获得许可。例如，编写使用本书中几个代码块的程序不需要许可。销售或者分发 O’Reilly 书籍的 CD-ROM 例子需要获得许可。

通过引用本书和使用示例代码来回答问题并不需要获得许可。将大量来自本书的示例代码整合到产品文档中并不需要获得许可。

我们感谢，但是并不要求，贡献。贡献通常包括标题，作者，出版商和 ISBN 。例如：“Hands-On Machine Learning withScikit-Learn and TensorFlow by Aurélien Géron (O’Reilly). Copyright 2017 AurélienGéron, 978-1-491-96229-9.”

如果您觉得您对代码示例的使用超出了合理使用范围或上述权限，请随时联系我们：permissions@oreilly.com 。

9、O’Reilly Safari

Safari （以前被称为 Safari Books Online）是一个针对企业，政府，教育工作者和个人的基于会员的培训和参考平台。

会员可以访问 250 多家发布商的数千本图书，培训视频，学习路径，互动教程和策划播放列表，其中包括 O’Reilly Media，哈佛商业评论，Prentice Hall 专业人员，Addison-Wesley 专业人员，Microsoft Press， Sams， Que， Peachpit Press， Adobe， Focal Press， Cisco Press 等。想要了解更多信息，请访问 http://oreilly.com/safari 。

10、如何联系我们

请向出版商发表有关本书的评论和问题：

O’Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 （在美国或者加拿大）

707-829-0515 （国际或地区）

707-829-0104 （传真）

我们有一个这本书的网页，在这里我们列出了勘误表，例子和任何额外的信息。你可以访问这个网页 http://bit.ly/hands-on-machine-learning-with-scikit-learn-and-tensorflow

要评论或者询问有关本书的技术问题，请发送电子邮件到 bookquestions@oreilly.com 。

有关我们的书籍，课程，会议和新闻的更多信息，请访问我们的网站 http://www.oreilly.com 。

在 facebook 上找到我们： http://facebook.com/oreilly

在 Twitter 上关注我们：http://twitter.com/oreillymedia

在 Youtube 上观看我们的视频： http://www.youtube.com/oreillymedia

11、致谢

我要感谢我的 Google 同事，特别是 Youtube 视频分类小组，教给我很多关于机器学习的知识。没有他们，我永远无法开始这个项目。特别感谢我的个人 ML 专家：Clément Courbet, Julien Dubois, Mathias Kende, Daniel Kitachewsky, James Pack, Alexander Pak, Anosh Raj, Vitor Sessak, Wiktor Tomczak, Ingrid von Glehn, Rich Washington, 以及 Youtube Paris 的所有人。

我非常感谢所有那些从繁忙的生活中抽出时间来仔细阅读我的书的人。感谢 Pete Warden 回答了我所有的 TensorFlow 的问题，回顾第二部分，提供了许多有趣的见解，当然也成为了 TensorFlow 核心团队的一员。你一定想要看看他的 博客 ！非常感谢 Lukas Biewald 对第二部分的非常全面的审查：他毫不留情地尝试了所有的代码（并且发现了一些错误），做出了许多伟大的建议，而且他的热情是具有感染力的。你应该看看他的博客，和他的超酷的机器人！感谢 Justin Francis ，他也非常全面地审查了第二部分，特别是在第 16 章提到了错误并提供了很好的见解。你可以在 TensorFlow 上看到他的帖子！

也非常感谢 David Andrzejewski，他审查了第一部分，提供了非常有用的反馈意见，确定了不明确的部分并提出了改进建议。查看一下他的网页吧。感谢 Grégoire Mesnil，他审查了第二部分，并提供了非常有趣的关于神经网络的实用建议。感谢 Eddy Hung, Salim Sémaoune, Karim Matrah, Ingrid von Glehn,Iain Smears, 和 Vincent Guilbeau 对第一部分的审查和建议。我还要感谢我的岳父，前数学老师 Michel Tessier ，现在是 Anton Chekhov 的一名优秀翻译，帮助我在本书中提供了一些非常好的数学和符号，并且审查了线性代数 Jupyter notebook 。

当然，对我亲爱的弟弟说一个巨大的“谢谢”，他测试了每一行代码，几乎在每个部分都提供了反馈，并鼓励我从第一行到最后一行。爱你，我的兄弟。

非常感谢 O’Reilly 出色的员工，特别是 Nicole Tache ，他给出了深刻的反馈，并且总是开朗，鼓舞和乐于助人的。还要感谢 Marie Beaugureau, Ben Lorica, Mike Loukides, 和 Laurel Ruma 相信这个项目并帮助我确定其范围。感谢 Matt Hacker 和所有的 Atlasteam 回答了关于格式化，asciidoc 和 LaTeX 的所有技术团队问题，也感谢 Rachel Monaghan, Nick Adams, 和所有的制作团队进行了最终的审查和数百次的修正。

最后但也很重要的一点，我非常感谢我的爱妻 Emmanuelle 和三个非常棒的孩子，Alexandre, Rémi, 和 Gabrielle ，在这本书中写了很多，问了很多问题（谁说不能教 7 岁的孩子神经网络？），甚至帮我送饼干和咖啡。你还梦想得到什么呢？

第1章 机器学习概览

大多数人听到“机器学习”，往往会在脑海中勾勒出一个机器人：一个可靠的管家，或是一个可怕的终结者，这取决于你问的是谁。但是机器学习并不是未来的幻想，它已经来到我们身边了。事实上，一些特定领域已经应用机器学习几十年了，比如光学字符识别（Optical Character Recognition，OCR）。但是直到 1990 年代，第一个影响了数亿人的机器学习应用才真正成熟，它就是垃圾邮件过滤器（spam filter）。虽然并不是一个有自我意识的天网系统（Skynet），垃圾邮件过滤器从技术上是符合机器学习的（它可以很好地进行学习，用户几乎不用再标记某个邮件为垃圾邮件）。后来出现了更多的数以百计的机器学习产品，支撑了更多你经常使用的产品和功能，从推荐系统到语音识别。

机器学习的起点和终点分别是什么呢？确切的讲，机器进行学习是什么意思？如果我下载了一份维基百科的拷贝，我的电脑就真的学会了什么吗？它马上就变聪明了吗？在本章中，我们首先会澄清机器学习到底是什么，以及为什么你要使用它。

然后，在我们出发去探索机器学习新大陆之前，我们要观察下地图，以便知道这片大陆上的主要地区和最明显的地标：监督学习vs非监督学习，在线学习vs批量学习，基于实例vs基于模型学习。然后，我们会学习一个典型的机器学习项目的工作流程，讨论可能碰到的难点，以及如何评估和微调一个机器学习系统。

这一章介绍了大量每个数据科学家需要牢记在心的基础概念（和习语）。第一章只是概览（唯一不含有代码的一章），相当简单，但你要确保每一点都搞明白了，再继续进行学习本书其余章节。端起一杯咖啡，开始学习吧！

> 提示：如果你已经知道了机器学习的所有基础概念，可以直接翻到第2章。如果你不确认，可以尝试回答本章末尾列出的问题，然后再继续。

什么是机器学习？

机器学习是通过编程让计算机从数据中进行学习的科学（和艺术）。

下面是一个更广义的概念：

机器学习是让计算机具有学习的能力，无需进行明确编程。 —— 亚瑟·萨缪尔，1959

和一个工程性的概念：

计算机程序利用经验 E 学习任务 T，性能是 P，如果针对任务 T 的性能 P 随着经验 E 不断增长，则称为机器学习。 —— 汤姆·米切尔，1997

例如，你的垃圾邮件过滤器就是一个机器学习程序，它可以根据垃圾邮件（比如，用户标记的垃圾邮件）和普通邮件（非垃圾邮件，也称作 ham）学习标记垃圾邮件。用来进行学习的样例称作训练集。每个训练样例称作训练实例（或样本）。在这个例子中，任务 T 就是标记新邮件是否是垃圾邮件，经验E是训练数据，性能P需要定义：例如，可以使用正确分类的比例。这个性能指标称为准确率，通常用在分类任务中。

如果你下载了一份维基百科的拷贝，你的电脑虽然有了很多数据，但不会马上变得聪明起来。因此，这不是机器学习。

为什么使用机器学习？

思考一下，你会如何使用传统的编程技术写一个垃圾邮件过滤器（图 1-1）：

1. 你先观察下垃圾邮件一般都是什么样子。你可能注意到一些词或短语（比如 4U、credit card、free、amazing）在邮件主题中频繁出现，也许还注意到发件人名字、邮件正文的格式，等等。
2. 你为观察到的规律写了一个检测算法，如果检测到了这些规律，程序就会标记邮件为垃圾邮件。
3. 测试程序，重复第1步和第2步，直到满足要求。

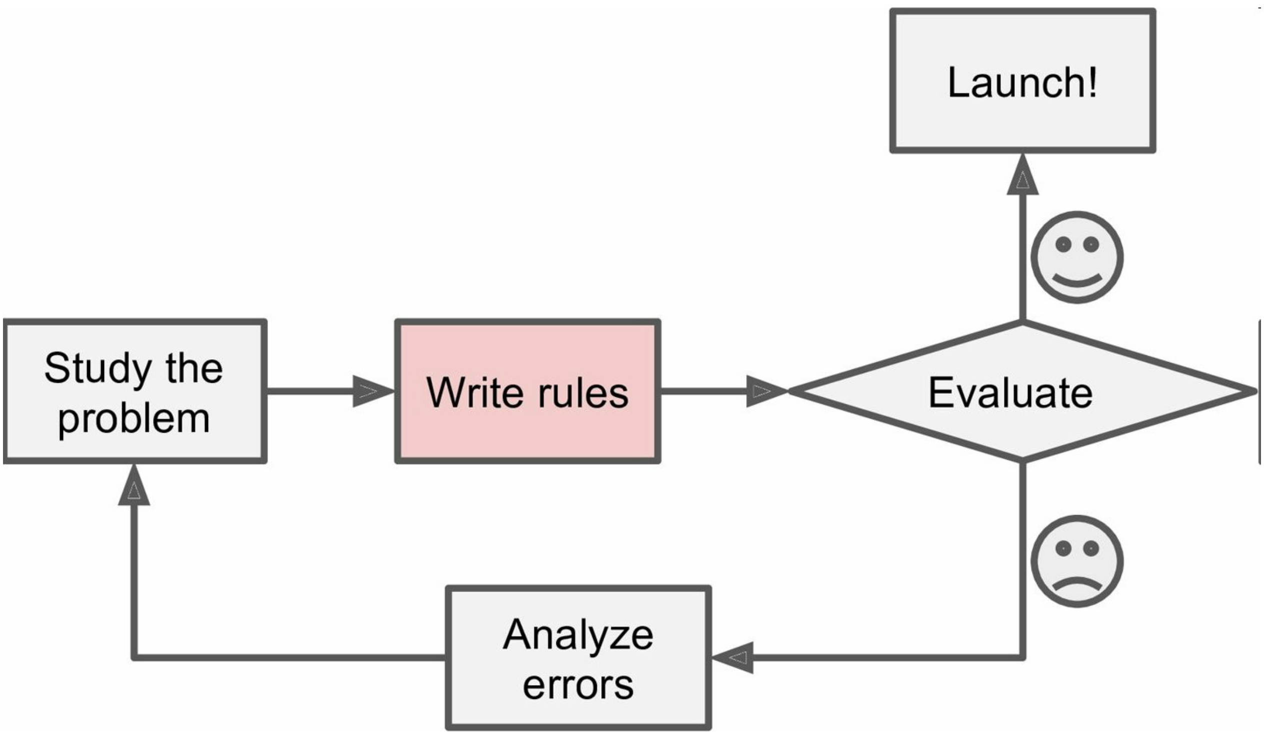


图 1-1 传统方法

这个问题并不简单，你的程序很可能会变成一长串复杂的规则—— 这样就会很难维护。

相反的，基于机器学习技术的垃圾邮件过滤器会自动学习哪个词和短语是垃圾邮件的预测值，通过与普通邮件比较，检测垃圾邮件中反常频次的词语格式（图 1-2）。这个程序短得多，更易维护，也更精确。

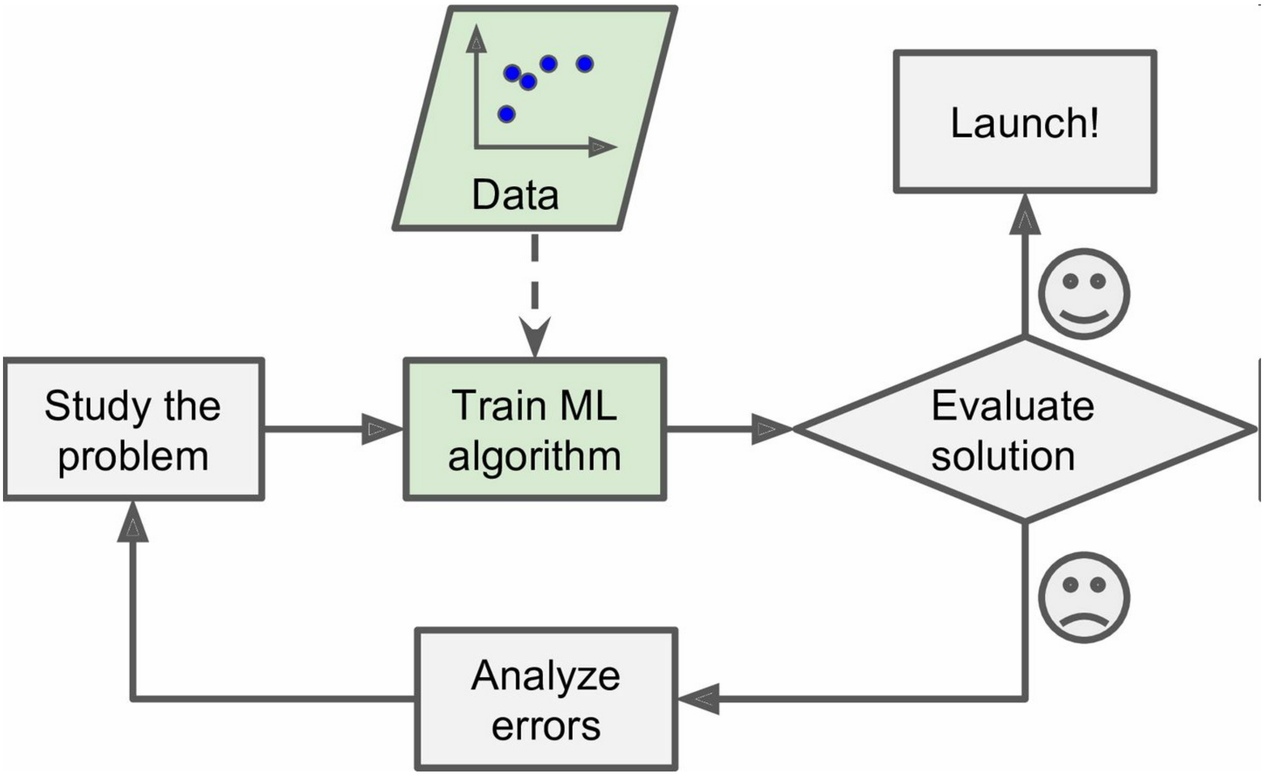


图 1-2 机器学习方法

进而，如果发送垃圾邮件的人发现所有包含“4U”的邮件都被屏蔽了，可能会转而使用“For U”。使用传统方法的垃圾邮件过滤器需要更新以标记“For U”。如果发送垃圾邮件的人持续更改，你就需要被动地不停地写入新规则。

相反的，基于机器学习的垃圾邮件过滤器会自动注意到“For U”在用户手动标记垃圾邮件中的反常频繁性，然后就能自动标记垃圾邮件而无需干预了（图1-3）。

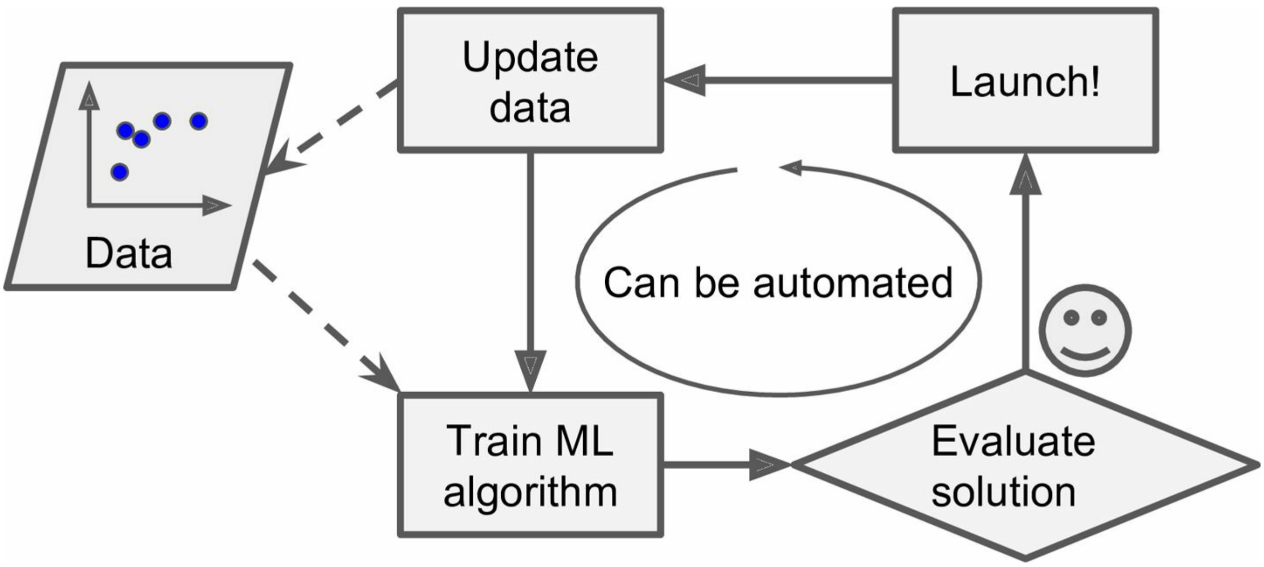


图 1-3 自动适应改变

机器学习的另一个优点是善于处理对于传统方法太复杂或是没有已知算法的问题。例如，对于语言识别：假如想写一个可以识别“one”和“two”的简单程序。你可能注意到“two”起始是一个高音（“T”），所以可以写一个可以测量高音强度的算法，用它区分 one 和 two。很明显，这个方法不能推广到嘈杂环境下的数百万人的数千词汇、数十种语言。（现在）最佳的方法是根据大量单词的录音，写一个可以自我学习的算法。

最后，机器学习可以帮助人类进行学习（图 1-4）：可以检查机器学习算法已经掌握了什么（尽管对于某些算法，这样做会有点麻烦）。例如，当垃圾邮件过滤器被训练了足够多的垃圾邮件，就可以用它列出垃圾邮件预测值的单词和单词组合列表。有时，可能会发现不引人关注的关联或新趋势，有助于对问题更好的理解。

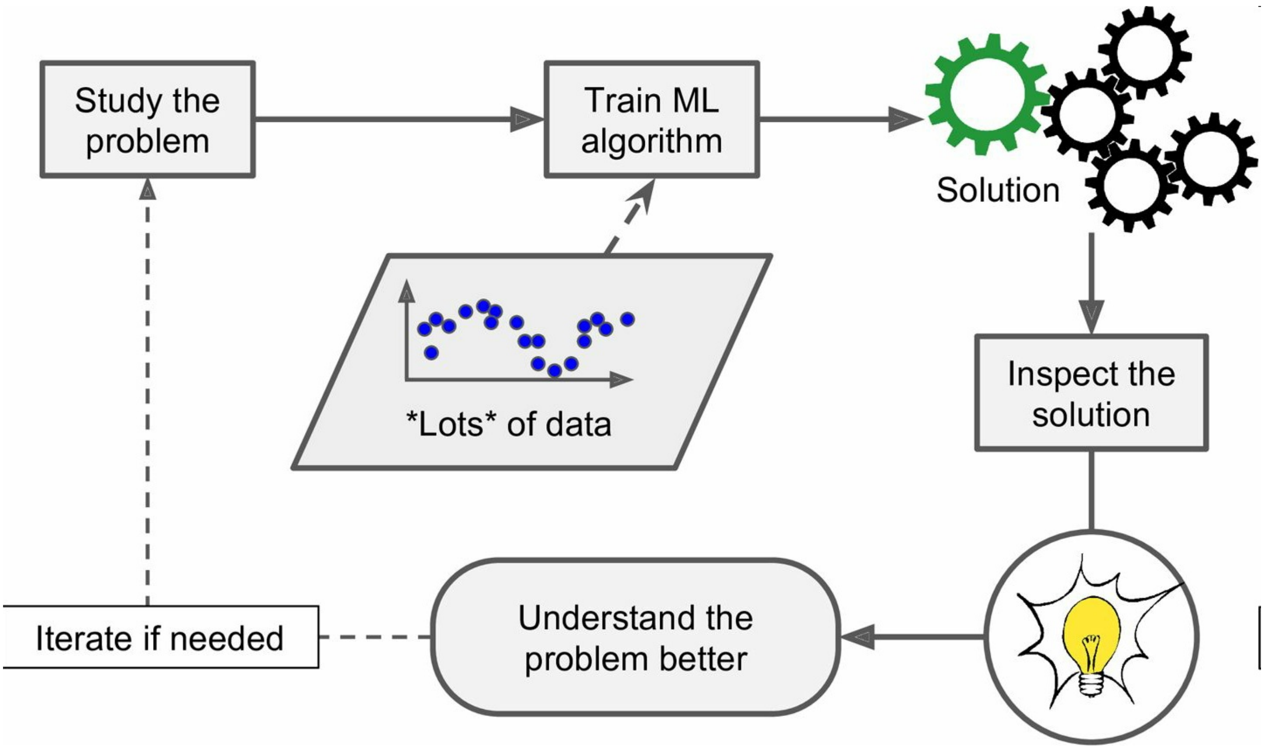


图 1-4 机器学习可以帮助人类学习

使用机器学习方法挖掘大量数据，可以发现并不显著的规律。这称作数据挖掘。

总结一下，机器学习善于：

- 需要进行大量手工调整或需要拥有长串规则才能解决的问题：机器学习算法通常可以简化代码、提高性能。
- 问题复杂，传统方法难以解决：最好的机器学习方法可以找到解决方案。
- 环境有波动：机器学习算法可以适应新数据。
- 洞察复杂问题和大量数据。

机器学习系统的类型

机器学习有多种类型，可以根据如下规则进行分类：

- 是否在人类监督下进行训练（监督，非监督，半监督和强化学习）
- 是否可以动态渐进学习（在线学习 vs 批量学习）
- 它们是否只是通过简单地比较新的数据点和已知的数据点，或者在训练数据中进行模式识别，以建立一个预测模型，就像科学家所做的那样（基于实例学习 vs 基于模型学习）

规则并不仅限于以上的，你可以将他们进行组合。例如，一个先进的垃圾邮件过滤器可以使用神经网络模型动态进行学习，用垃圾邮件和普通邮件进行训练。这就让它成了一个在线、基于模型、监督学习系统。

下面更仔细地学习这些规则。

监督/非监督学习

机器学习可以根据训练时监督的量和类型进行分类。主要有四类：监督学习、非监督学习、半监督学习和强化学习。

监督学习

在监督学习中，用来训练算法的训练数据包含了答案，称为标签（图 1-5）。

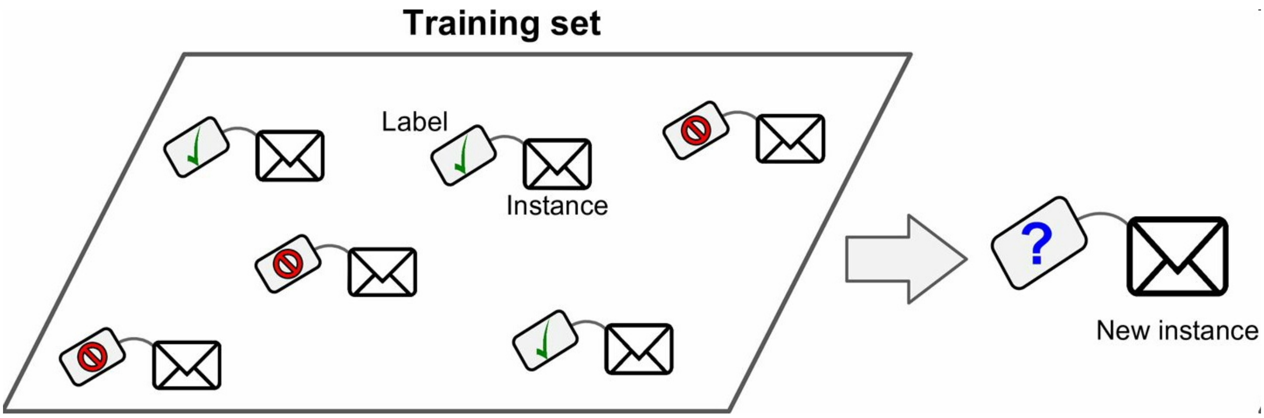


图 1-5 用于监督学习（比如垃圾邮件分类）的加了标签的训练集

一个典型的监督学习任务是分类。垃圾邮件过滤器就是一个很好的例子：用许多带有归类（垃圾邮件或普通邮件）的邮件样本进行训练，过滤器必须还能对新邮件进行分类。

另一个典型任务是预测目标数值，例如给出一些特征（里程数、车龄、品牌等等）称作预测值，来预测一辆汽车的价格。这类任务称作回归（图 1-6）。要训练这个系统，你需要给出大量汽车样本，包括它们的预测值和标签（即，它们的价格）。

> 注解：在机器学习中，一个属性就是一个数据类型（例如，“里程数”），取决于具体问题一个特征会有多个含义，但通常是属性加上它的值（例如，“里程数`=15000`”）。许多人是不区分地使用属性和特征。

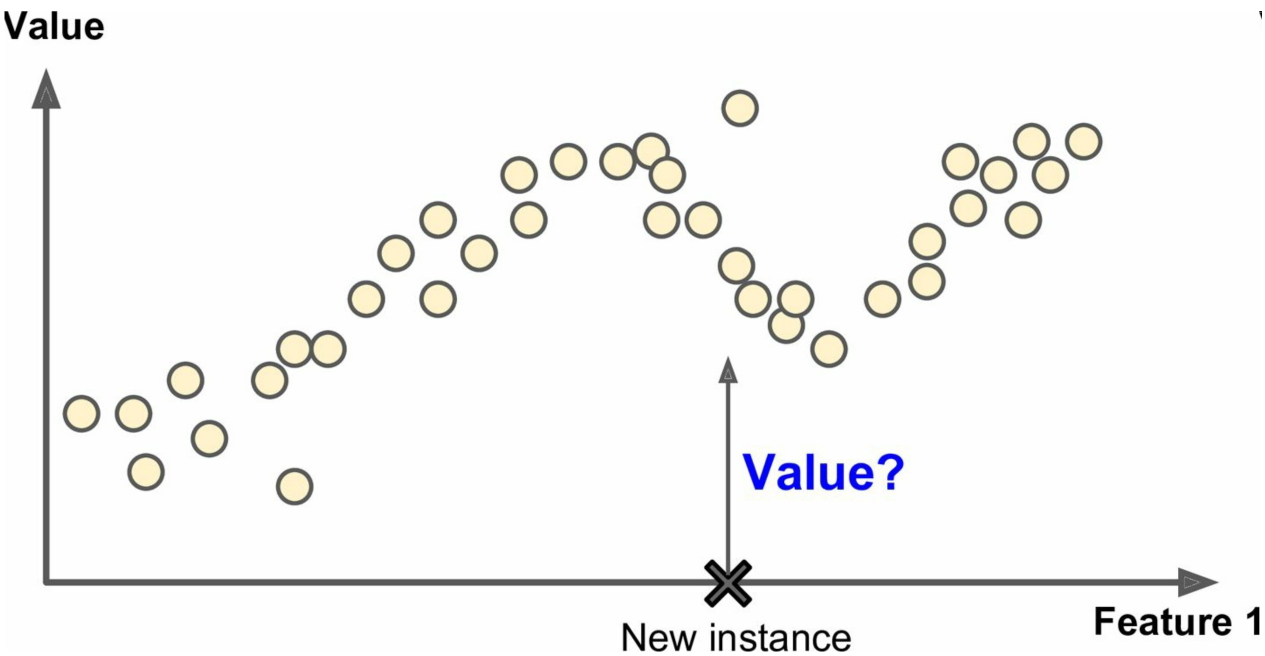


图 1-6 回归

注意，一些回归算法也可以用来进行分类，反之亦然。例如，逻辑回归通常用来进行分类，它可以生成一个归属某一类的可能性的值（例如，20% 几率为垃圾邮件）。

下面是一些重要的监督学习算法（本书都有介绍）：

- K近邻算法
- 线性回归
- 逻辑回归
- 支持向量机（SVM）
- 决策树和随机森林
- 神经网络

非监督学习

在非监督学习中，你可能猜到了，训练数据是没有加标签的（图 1-7）。系统在没有老师的条件下进行学习。

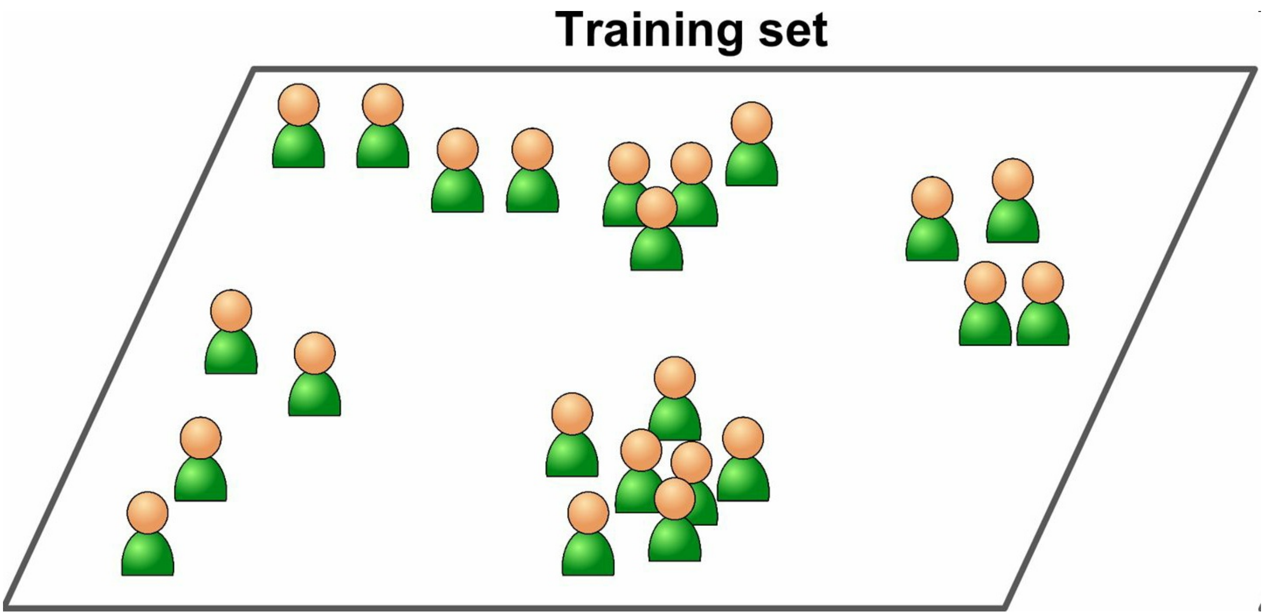


图 1-7 非监督学习的一个不加标签的训练集

下面是一些最重要的非监督学习算法（我们会在第 8 章介绍降维）：

- 聚类
  K 均值
  层次聚类分析（Hierarchical Cluster Analysis，HCA）
  期望最大值
- 可视化和降维
  主成分分析（Principal Component Analysis，PCA）
  核主成分分析
  局部线性嵌入（Locally-Linear Embedding，LLE）
  t-分布邻域嵌入算法（t-distributed Stochastic Neighbor Embedding，t-SNE）
- 关联性规则学习
  Apriori 算法
  Eclat 算法

例如，假设你有一份关于你的博客访客的大量数据。你想运行一个聚类算法，检测相似访客的分组（图 1-8）。你不会告诉算法某个访客属于哪一类：它会自己找出关系，无需帮助。例如，算法可能注意到 40% 的访客是喜欢漫画书的男性，通常是晚上访问，20% 是科幻爱好者，他们是在周末访问等等。如果你使用层次聚类分析，它可能还会细分每个分组为更小的组。这可以帮助你为每个分组定位博文。

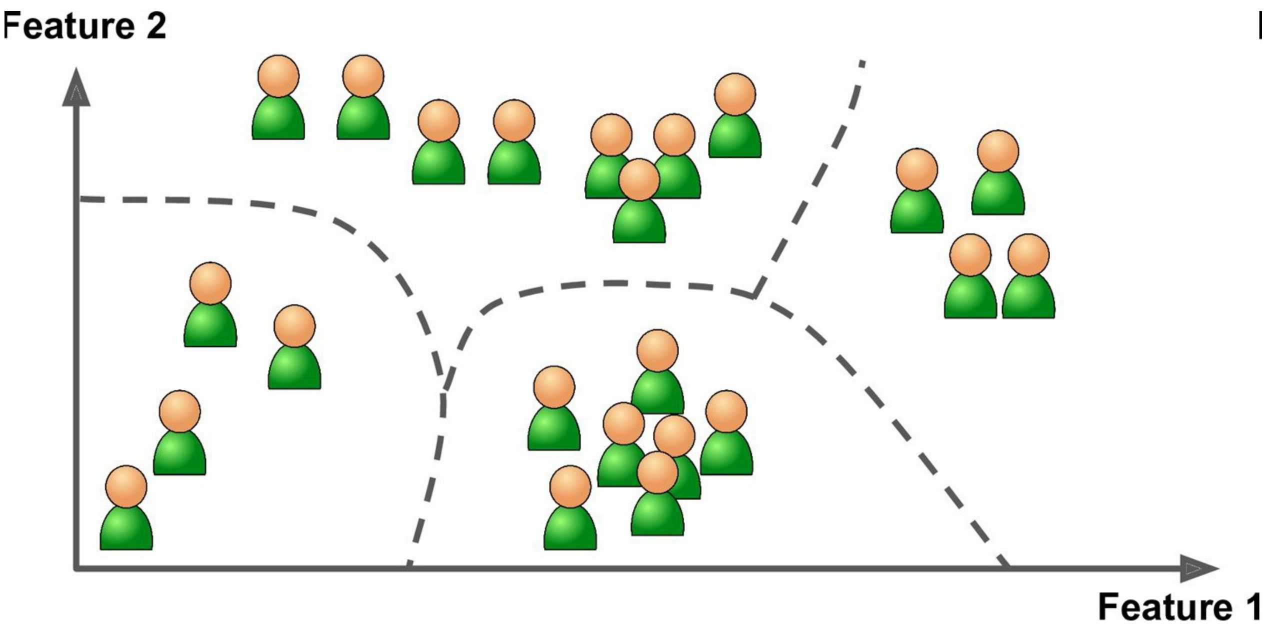


图 1-8 聚类

可视化算法也是极佳的非监督学习案例：给算法大量复杂的且不加标签的数据，算法输出数据的2D或3D图像（图 1-9）。算法会试图保留数据的结构（即尝试保留输入的独立聚类，避免在图像中重叠），这样就可以明白数据是如何组织起来的，也许还能发现隐藏的规律。

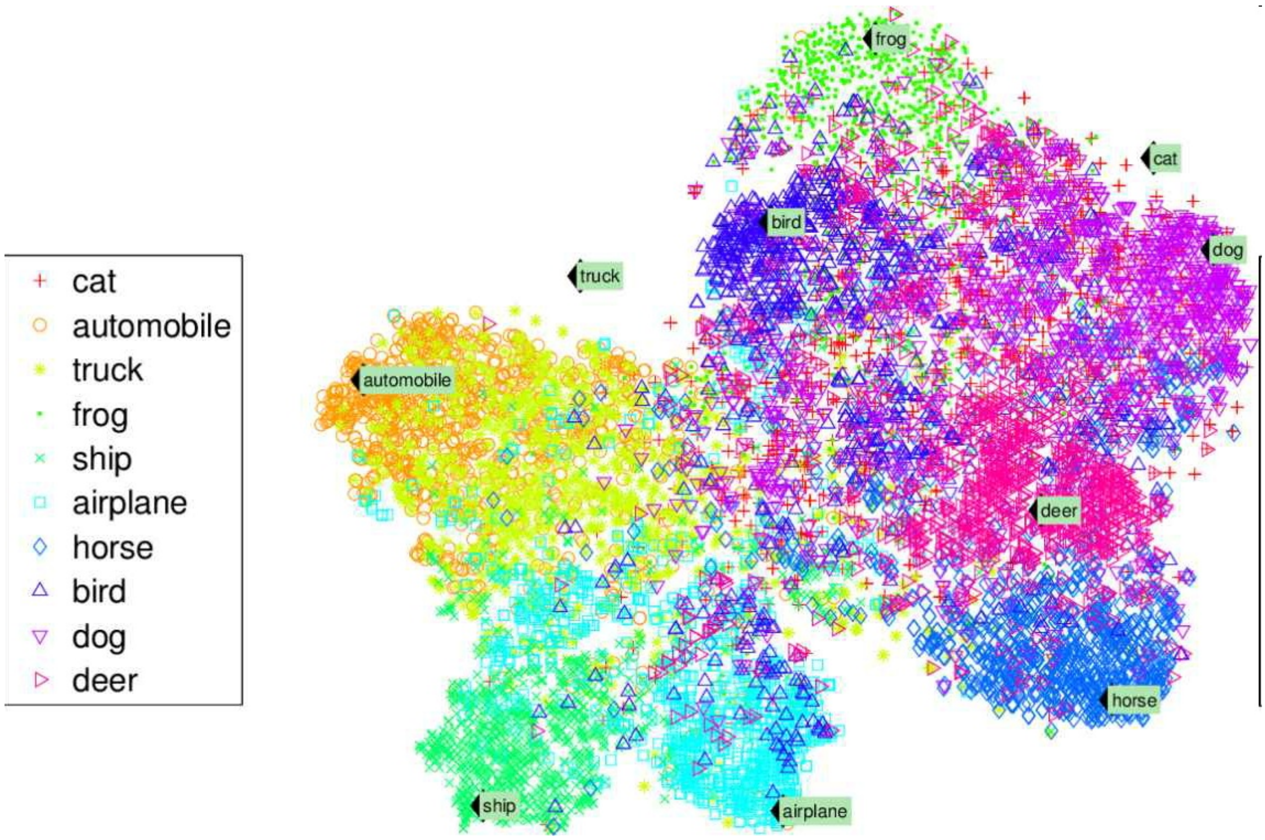


图 1-9 t-SNE 可视化案例，突出了聚类（注：注意动物是与汽车分开的，马和鹿很近、与鸟距离远，以此类推）

与此有关联的任务是降维，降维的目的是简化数据、但是不能失去大部分信息。做法之一是合并若干相关的特征。例如，汽车的里程数与车龄高度相关，降维算法就会将它们合并成一个，表示汽车的磨损。这叫做特征提取。

> 提示：在用训练集训练机器学习算法（比如监督学习算法）时，最好对训练集进行降维。这样可以运行的更快，占用的硬盘和内存空间更少，有些情况下性能也更好。

另一个重要的非监督任务是异常检测（anomaly detection）——例如，检测异常的信用卡转账以防欺诈，检测制造缺陷，或者在训练之前自动从训练数据集去除异常值。异常检测的系统使用正常值训练的，当它碰到一个新实例，它可以判断这个新实例是像正常值还是异常值（图 1-10）。



图 1-10 异常检测

最后，另一个常见的非监督任务是关联规则学习，它的目标是挖掘大量数据以发现属性间有趣的关系。例如，假设你拥有一个超市。在销售日志上运行关联规则，可能发现买了烧烤酱和薯片的人也会买牛排。因此，你可以将这些商品放在一起。

半监督学习

一些算法可以处理部分带标签的训练数据，通常是大量不带标签数据加上小部分带标签数据。这称作半监督学习（图 1-11）。

一些图片存储服务，比如 Google Photos，是半监督学习的好例子。一旦你上传了所有家庭相片，它就能自动识别相同的人 A 出现了相片 1、5、11 中，另一个人 B 出现在了相片 2、5、7 中。这是算法的非监督部分（聚类）。现在系统需要的就是你告诉这两个人是谁。只要给每个人一个标签，算法就可以命名每张照片中的每个人，特别适合搜索照片。

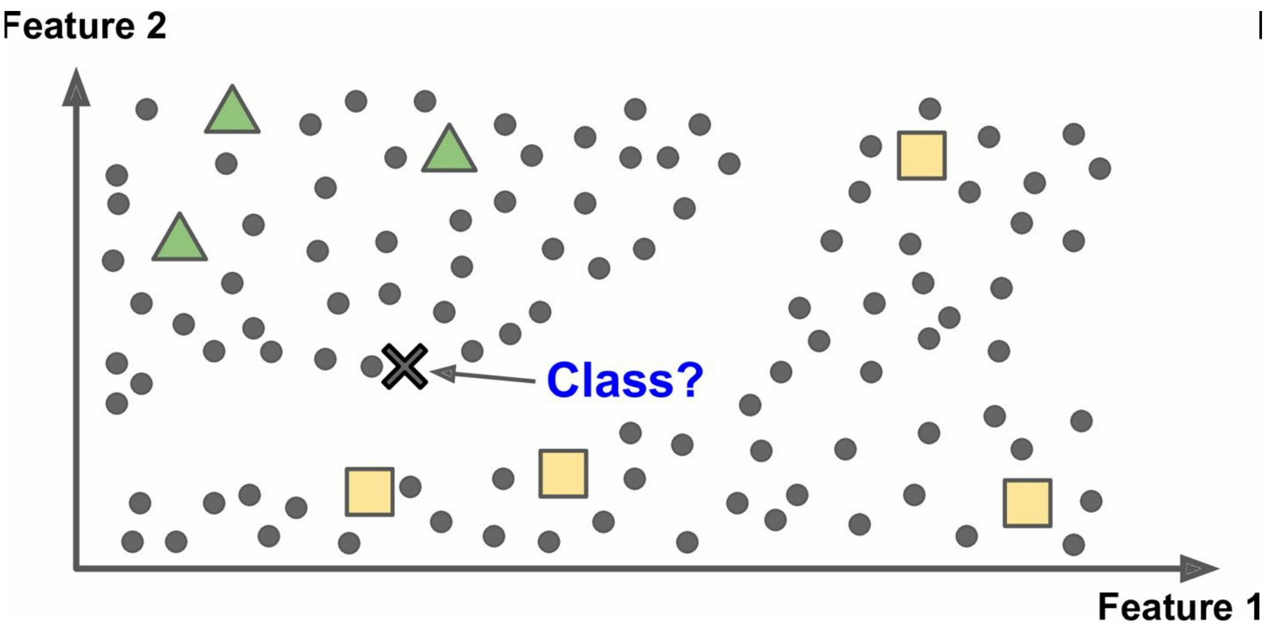


图 1-11 半监督学习

多数半监督学习算法是非监督和监督算法的结合。例如，深度信念网络（deep belief networks）是基于被称为互相叠加的受限玻尔兹曼机（restricted Boltzmann machines，RBM）的非监督组件。RBM 是先用非监督方法进行训练，再用监督学习方法进行整个系统微调。

强化学习

强化学习非常不同。学习系统在这里被称为智能体（agent），可以对环境进行观察，选择和执行动作，获得奖励（负奖励是惩罚，见图 1-12）。然后它必须自己学习哪个是最佳方法（称为策略，policy），以得到长久的最大奖励。策略决定了智能体在给定情况下应该采取的行动。

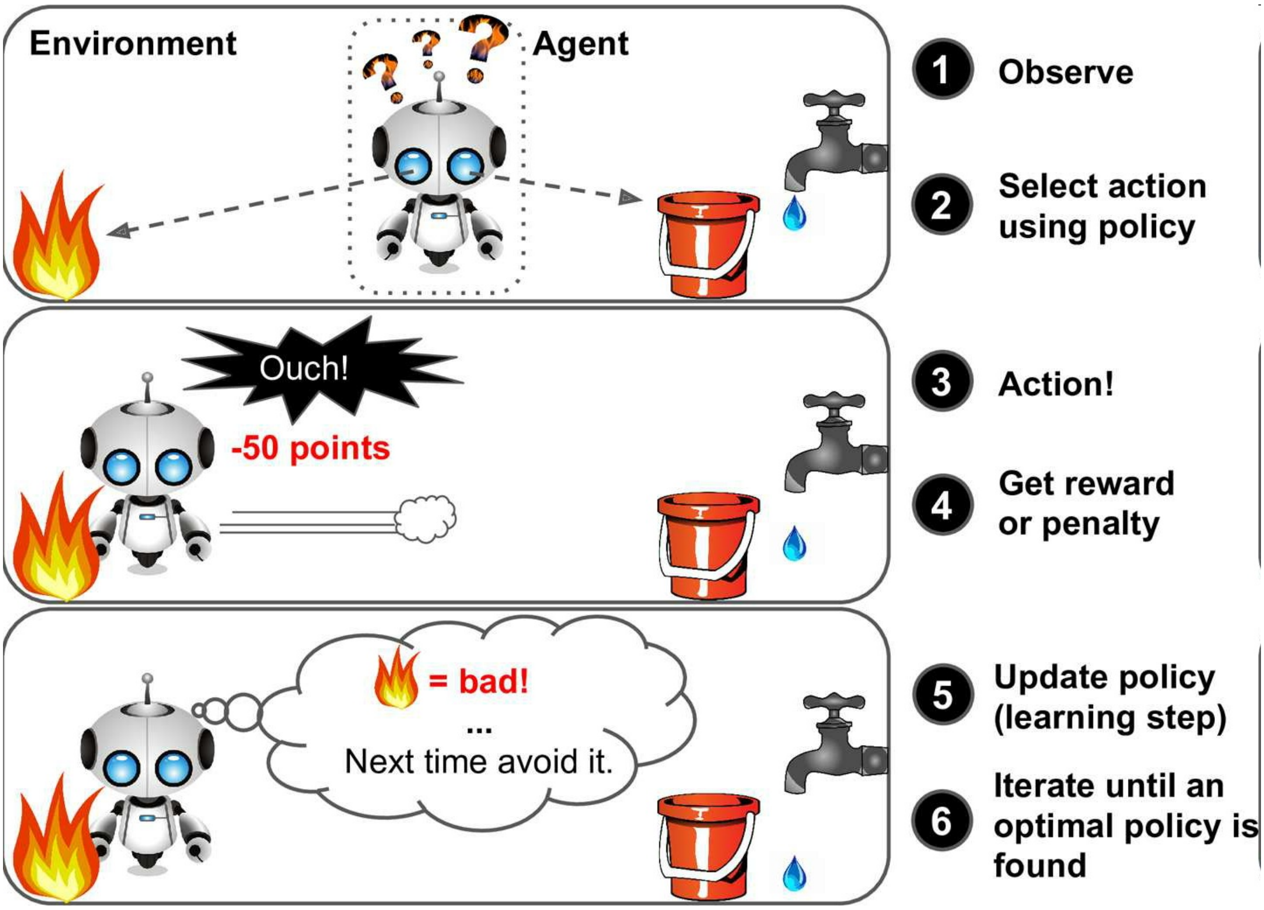


图 1-12 强化学习

例如，许多机器人运行强化学习算法以学习如何行走。DeepMind 的 AlphaGo 也是强化学习的例子：它在 2016 年三月击败了世界围棋冠军李世石（译者注：2017 年五月，AlphaGo 又击败了世界排名第一的柯洁）。它是通过分析数百万盘棋局学习制胜策略，然后自己和自己下棋。要注意，在比赛中机器学习是关闭的；AlphaGo 只是使用它学会的策略。

批量和在线学习

另一个用来分类机器学习的准则是，它是否能从导入的数据流进行持续学习。

批量学习

在批量学习中，系统不能进行持续学习：必须用所有可用数据进行训练。这通常会占用大量时间和计算资源，所以一般是线下做的。首先是进行训练，然后部署在生产环境且停止学习，它只是使用已经学到的策略。这称为离线学习。

如果你想让一个批量学习系统明白新数据（例如垃圾邮件的新类型），就需要从头训练一个系统的新版本，使用全部数据集（不仅有新数据也有老数据），然后停掉老系统，换上新系统。

幸运的是，训练、评估、部署一套机器学习的系统的整个过程可以自动进行（见图 1-3），所以即便是批量学习也可以适应改变。只要有需要，就可以方便地更新数据、训练一个新版本。

这个方法很简单，通常可以满足需求，但是用全部数据集进行训练会花费大量时间，所以一般是每 24 小时或每周训练一个新系统。如果系统需要快速适应变化的数据（比如，预测股价变化），就需要一个响应更及时的方案。

另外，用全部数据训练需要大量计算资源（CPU、内存空间、磁盘空间、磁盘 I/O、网络 I/O 等等）。如果你有大量数据，并让系统每天自动从头开始训练，就会开销很大。如果数据量巨大，甚至无法使用批量学习算法。

最后，如果你的系统需要自动学习，但是资源有限（比如，一台智能手机或火星车），携带大量训练数据、每天花费数小时的大量资源进行训练是不实际的。

幸运的是，对于上面这些情况，还有一个更佳的方案可以进行持续学习。

在线学习

在在线学习中，是用数据实例持续地进行训练，可以一次一个或一次几个实例（称为小批量）。每个学习步骤都很快且廉价，所以系统可以动态地学习到达的新数据（见图 1-13）。

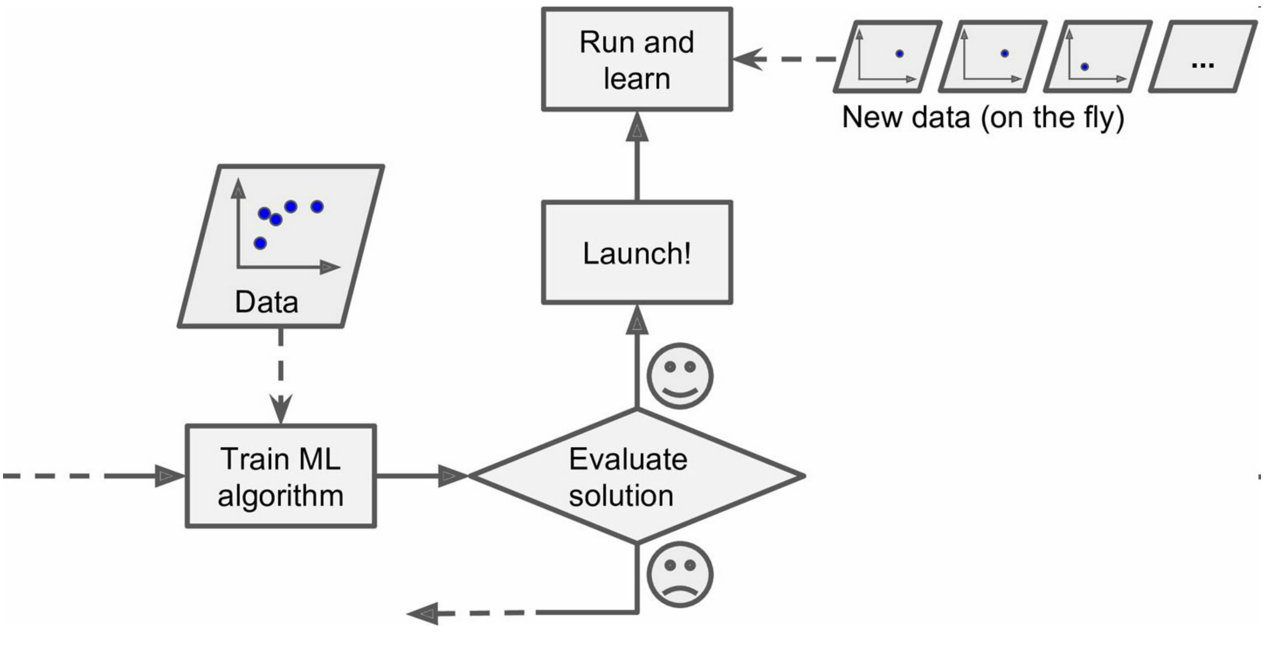


图 1-13 在线学习

在线学习很适合系统接收连续流的数据（比如，股票价格），且需要自动对改变作出调整。如果计算资源有限，在线学习是一个不错的方案：一旦在线学习系统学习了新的数据实例，它就不再需要这些数据了，所以扔掉这些数据（除非你想滚回到之前的一个状态，再次使用数据）。这样可以节省大量的空间。

在线学习算法也可以当机器的内存存不下大量数据集时，用来训练系统（这称作核外学习，*out-of-core* learning）。算法加载部分的数据，用这些数据进行训练，重复这个过程，直到用所有数据都进行了训练（见图 1-14）。

> 警告：这个整个过程通常是离线完成的（即，不在部署的系统上），所以在线学习这个名字会让人疑惑。可以把它想成持续学习。

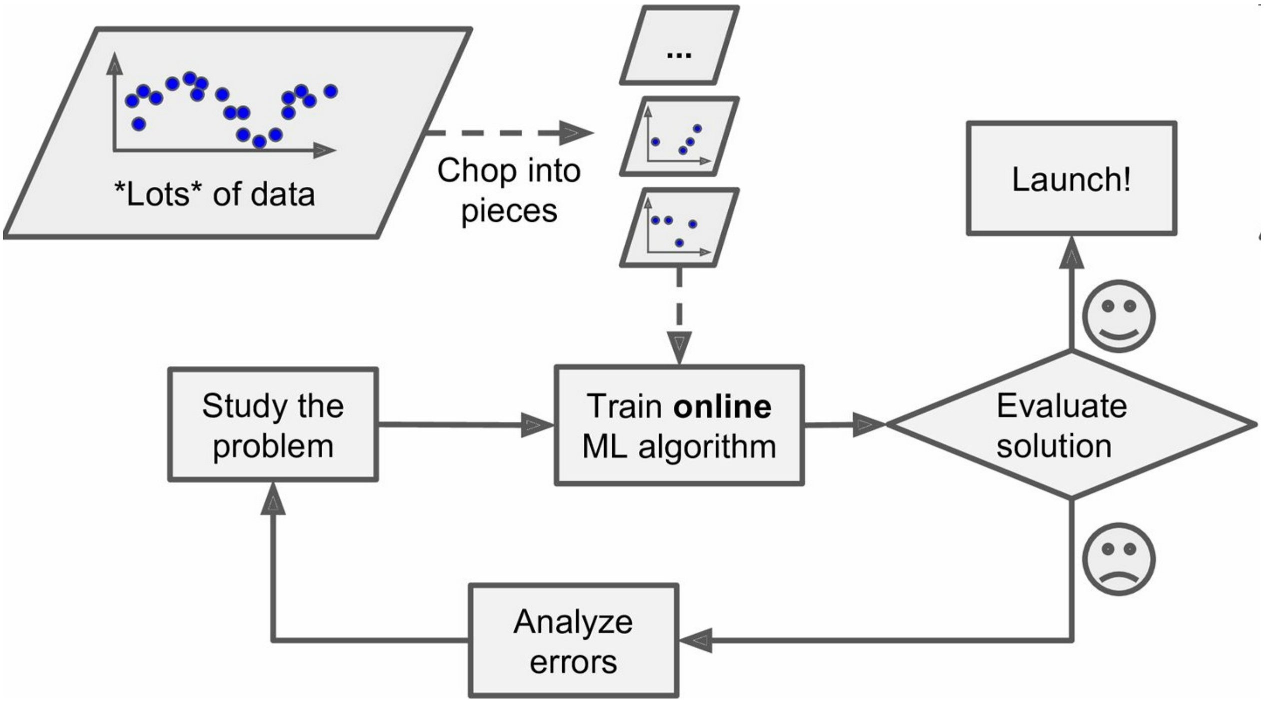


图 1-14 使用在线学习处理大量数据集

在线学习系统的一个重要参数是，它们可以多快地适应数据的改变：这被称为学习速率。如果你设定一个高学习速率，系统就可以快速适应新数据，但是也会快速忘记老数据（你可不想让垃圾邮件过滤器只标记最新的垃圾邮件种类）。相反的，如果你设定的学习速率低，系统的惰性就会强：即，它学的更慢，但对新数据中的噪声或没有代表性的数据点结果不那么敏感。

在线学习的挑战之一是，如果坏数据被用来进行训练，系统的性能就会逐渐下滑。如果这是一个部署的系统，用户就会注意到。例如，坏数据可能来自失灵的传感器或机器人，或某人向搜索引擎传入垃圾信息以提高搜索排名。要减小这种风险，你需要密集监测，如果检测到性能下降，要快速关闭（或是滚回到一个之前的状态）。你可能还要监测输入数据，对反常数据做出反应（比如，使用异常检测算法）。

基于实例 vs 基于模型学习

另一种分类机器学习的方法是判断它们是如何进行归纳推广的。大多机器学习任务是关于预测的。这意味着给定一定数量的训练样本，系统需要能推广到之前没见到过的样本。对训练数据集有很好的性能还不够，真正的目标是对新实例预测的性能。

有两种主要的归纳方法：基于实例学习和基于模型学习。

基于实例学习

也许最简单的学习形式就是用记忆学习。如果用这种方法做一个垃圾邮件检测器，只需标记所有和用户标记的垃圾邮件相同的邮件 —— 这个方法不差，但肯定不是最好的。

不仅能标记和已知的垃圾邮件相同的邮件，你的垃圾邮件过滤器也要能标记类似垃圾邮件的邮件。这就需要测量两封邮件的相似性。一个（简单的）相似度测量方法是统计两封邮件包含的相同单词的数量。如果一封邮件含有许多垃圾邮件中的词，就会被标记为垃圾邮件。

这被称作基于实例学习：系统先用记忆学习案例，然后使用相似度测量推广到新的例子（图1-15）。

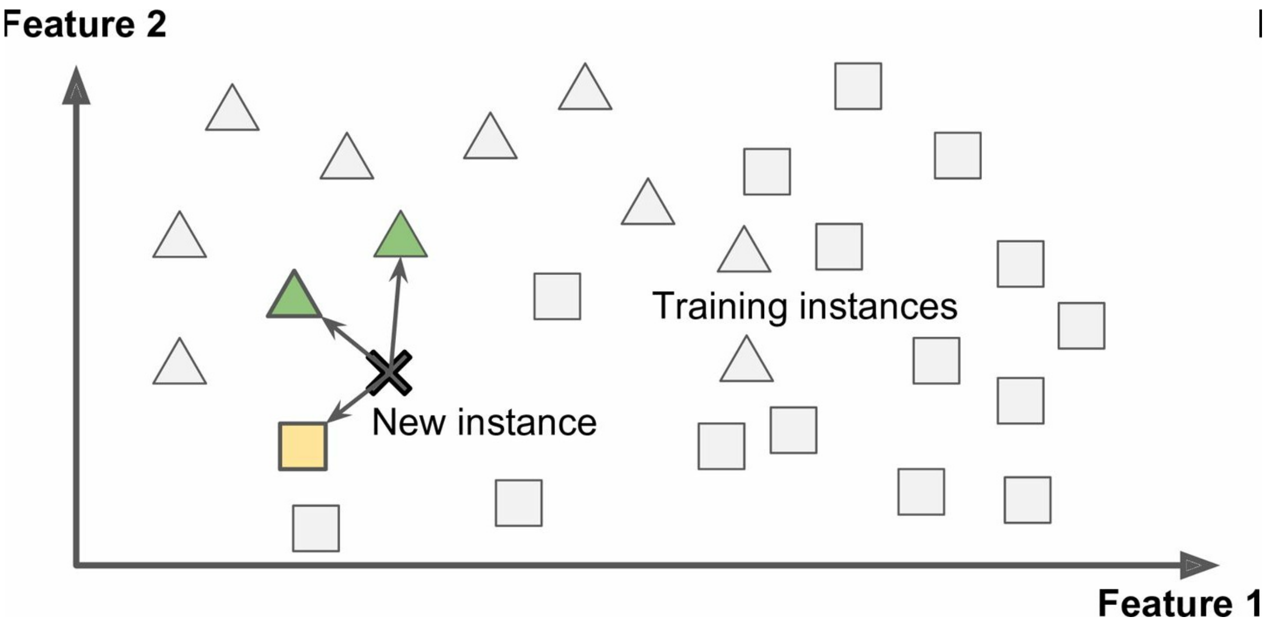


图 1-15 基于实例学习

基于模型学习

另一种从样本集进行归纳的方法是建立这些样本的模型，然后使用这个模型进行预测。这称作基于模型学习（图 1-16）。

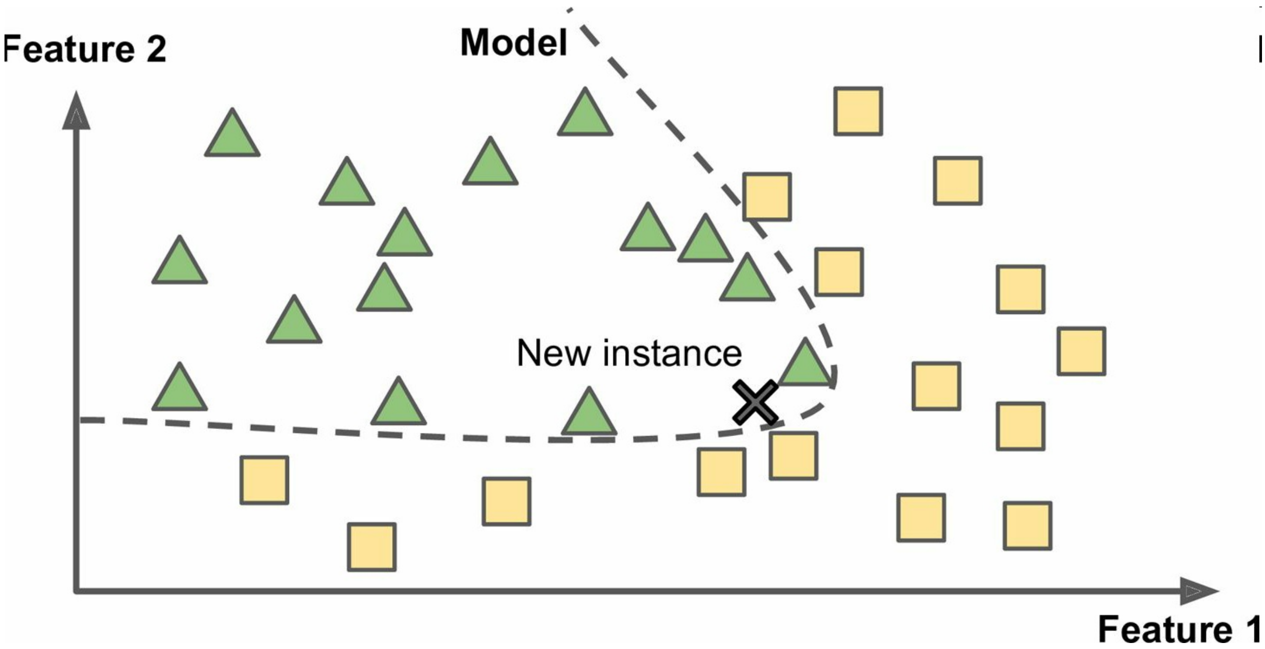


图 1-16 基于模型学习

例如，你想知道钱是否能让人快乐，你从 OECD 网站下载了 Better Life Index 指数数据，还从 IMF 下载了人均 GDP 数据。表 1-1 展示了摘要。

国家	人均 GDP（美元）	生活满意度
匈牙利	12240	4.9
韩国	27195	5.8
发过	37675	6.5
澳大利亚	50962	7.3
美国	55805	7.2

表 1-1 钱会使人幸福吗？

用一些国家的数据画图（图 1-17）。

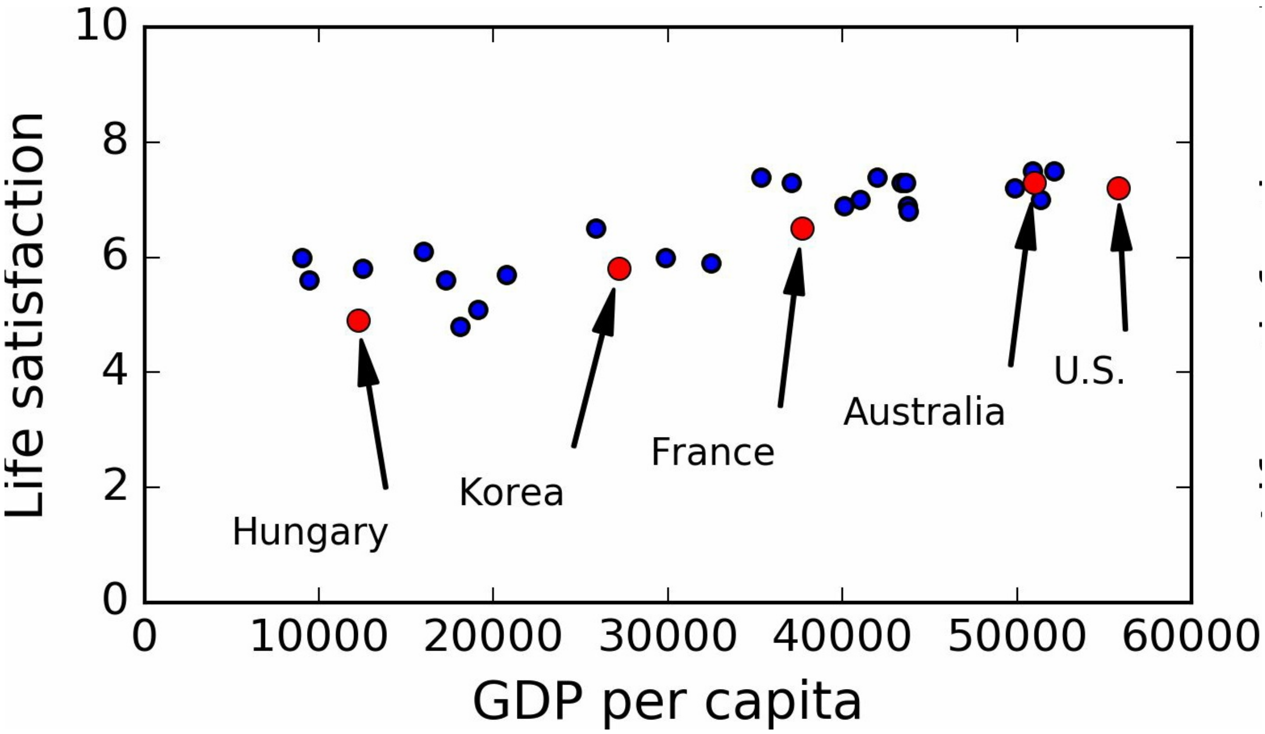


图 1-17 你看到趋势了吗？

确实能看到趋势！尽管数据有噪声（即，部分随机），看起来生活满意度是随着人均 GDP 的增长线性提高的。所以，你决定生活满意度建模为人均 GDP 的线性函数。这一步称作模型选择：你选一个生活满意度的线性模型，只有一个属性，人均 GDP（公式 1-1）。

$$life_satisfaction = \theta_0 + \theta_1 \times GDP_per_capita$$

公式 1-1 一个简单的线性模型

这个模型有两个参数 `θ0` 和 `θ1` 。通过调整这两个参数，你可以使你的模型表示任何线性函数，见图 1-18。

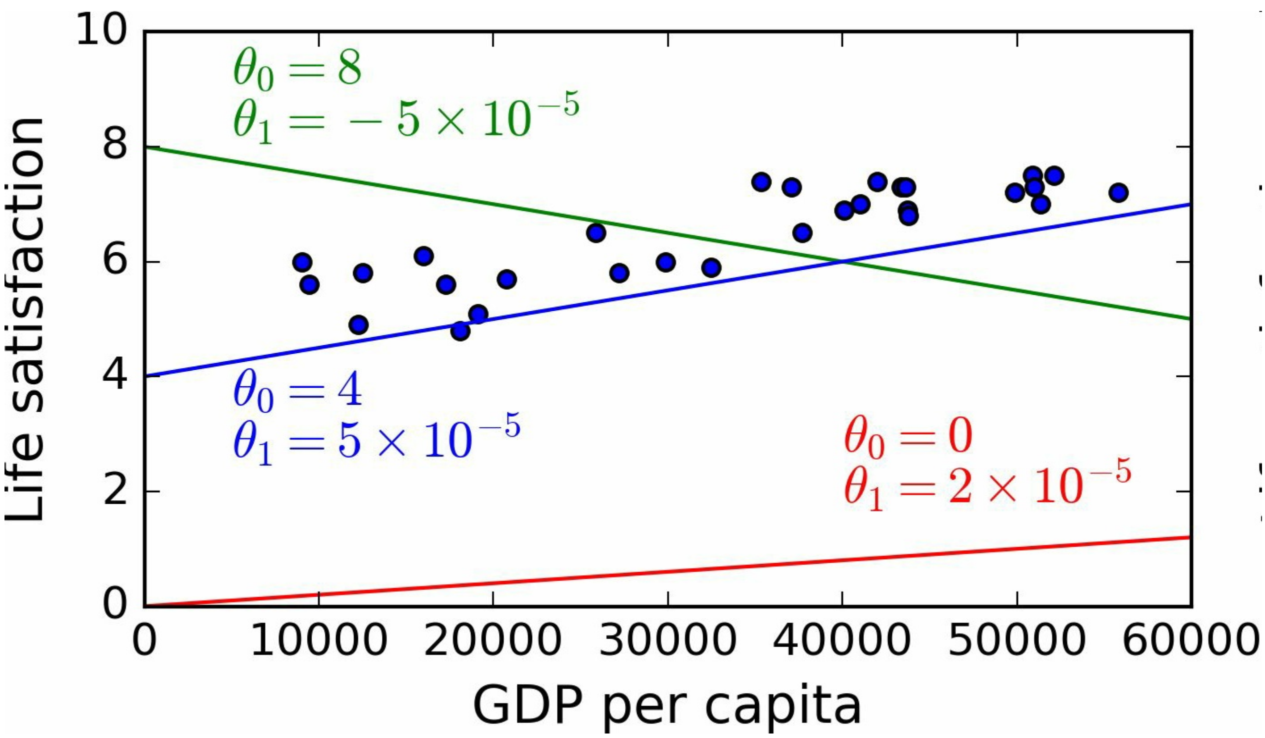


图 1-18 几个可能的线性模型

在使用模型之前，你需要确定 `θ0` 和 `θ1` 。如何能知道哪个值可以使模型的性能最佳呢？要回答这个问题，你需要指定性能的量度。你可以定义一个实用函数（或拟合函数）用来测量模型是否够好，或者你可以定义一个代价函数来测量模型有多差。对于线性回归问题，人们一般是用代价函数测量线性模型的预测值和训练样本的距离差，目标是使距离差最小。

接下来就是线性回归算法，你用训练样本训练算法，算法找到使线性模型最拟合数据的参数。这称作模型训练。在我们的例子中，算法得到的参数值是 `θ0=4.85` 和 `θ1=4.91×10-5` 。

现在模型已经最紧密地拟合到训练数据了，见图 1-19。

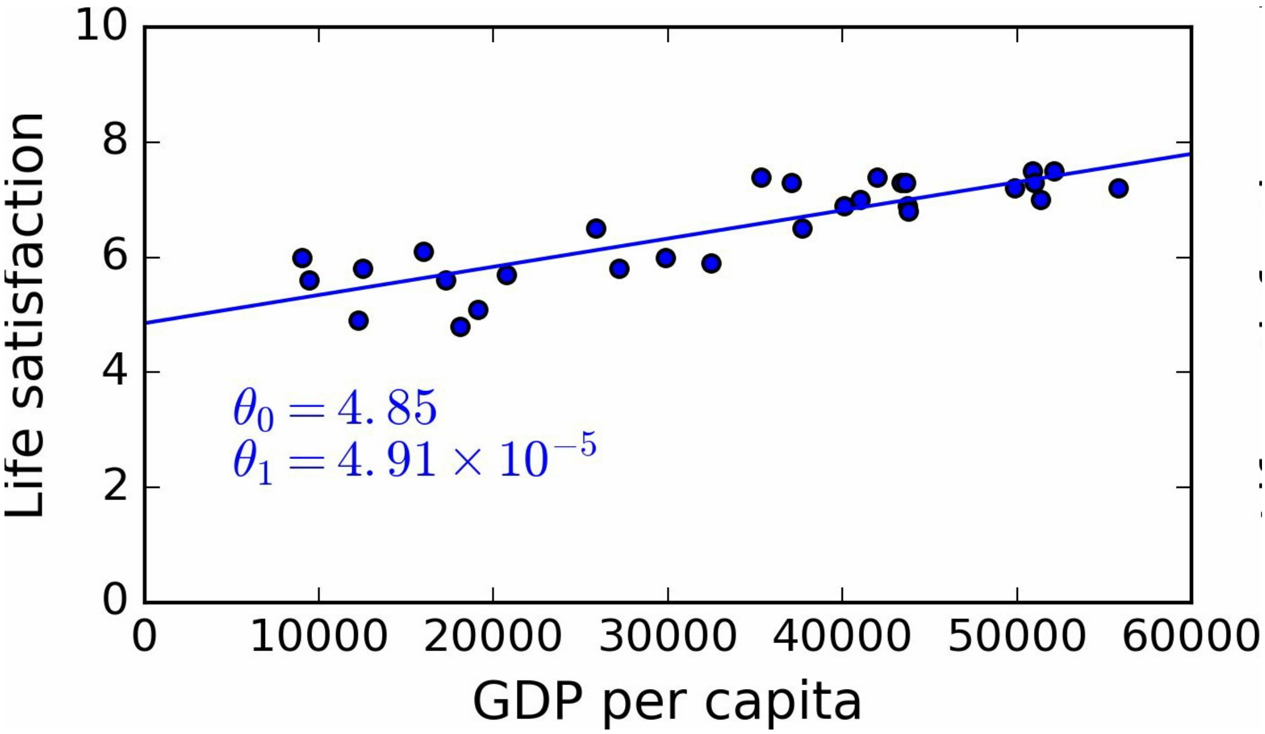


图 1-19 最佳拟合训练数据的线性模型

最后，可以准备运行模型进行预测了。例如，假如你想知道塞浦路斯人有多幸福，但 OECD 没有它的数据。幸运的是，你可以用模型进行预测：查询塞浦路斯的人均 GDP，为 22587 美元，然后应用模型得到生活满意度，后者的值在 `4.85 + 22,587 × 4.91 × 10-5 = 5.96` 左右。

为了激起你的兴趣，案例 1-1 展示了加载数据、准备、创建散点图的 Python 代码，然后训练线性模型并进行预测。

案例 1-1，使用 Scikit-Learn 训练并运行线性模型。

```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sklearn

# 加载数据
oecd_bli = pd.read_csv("oecd_bli_2015.csv", thousands=',')
gdp_per_capita = pd.read_csv("gdp_per_capita.csv",thousands=',',delimiter='\t',
                             encoding='latin1', na_values="n/a")

# 准备数据
country_stats = prepare_country_stats(oecd_bli, gdp_per_capita)
X = np.c_[country_stats["GDP per capita"]]
y = np.c_[country_stats["Life satisfaction"]]

# 可视化数据
country_stats.plot(kind='scatter', x="GDP per capita", y='Life satisfaction')
plt.show()

# 选择线性模型
lin_reg_model = sklearn.linear_model.LinearRegression()

# 训练模型
lin_reg_model.fit(X, y)

# 对塞浦路斯进行预测
X_new = [[22587]]  # 塞浦路斯的人均GDP
print(lin_reg_model.predict(X_new)) # outputs [[ 5.96242338]]
```

> 注解：如果你之前接触过基于实例学习算法，你会发现斯洛文尼亚的人均 GDP（20732 美元）和塞浦路斯差距很小，OECD 数据上斯洛文尼亚的生活满意度是 5.7，就可以预测塞浦路斯的生活满意度也是 5.7。如果放大一下范围，看一下接下来两个临近的国家，你会发现葡萄牙和西班牙的生活满意度分别是 5.1 和 6.5。对这三个值进行平均得到 5.77，就和基于模型的预测值很接近。这个简单的算法叫做k近邻回归（这个例子中，`k=3`）。
>
> 在前面的代码中替换线性回归模型为 K 近邻模型，只需更换下面一行：
>
> ```
> clf = sklearn.linear_model.LinearRegression()
> ```
>
> 为：
>
> ```
> clf = sklearn.neighbors.KNeighborsRegressor(n_neighbors=3)
> ```

如果一切顺利，你的模型就可以作出好的预测。如果不能，你可能需要使用更多的属性（就业率、健康、空气污染等等），获取更多更好的训练数据，或选择一个更好的模型（比如，多项式回归模型）。

总结一下：

- 研究数据
- 选择模型
- 用训练数据进行训练（即，学习算法搜寻模型参数值，使代价函数最小）

- 最后，使用模型对新案例进行预测（这称作推断），但愿这个模型推广效果不差

这就是一个典型的机器学习项目。在第 2 章中，你会第一手地接触一个完整的项目。

我们已经学习了许多关于基础的内容：你现在知道了机器学习是关于什么的，为什么它这么有用，最常见的机器学习的分类，典型的项目工作流程。现在，让我们看一看学习中会发生什么错误，导致不能做出准确的预测。

机器学习的主要挑战

简而言之，因为你的主要任务是选择一个学习算法并用一些数据进行训练，会导致错误的两件事就是“错误的算法”和“错误的数据”。我们从错误的数据开始。

训练数据量不足

要让一个蹒跚学步的孩子知道什么是苹果，需要做的就是指着一个苹果说“苹果”（可能需要重复这个过程几次）。现在这个孩子就能认识所有形状和颜色的苹果。真是个天才！

机器学习还达不到这个程度；需要大量数据，才能让多数机器学习算法正常工作。即便对于非常简单的问题，一般也需要数千的样本，对于复杂的问题，比如图像或语音识别，你可能需要数百万的样本（除非你能重复使用部分存在的模型）。

数据不合理的有效性

在一篇 2001 年发表的著名论文中，微软研究员 Michele Banko 和 Eric Brill 展示了不同的机器学习算法，包括非常简单的算法，一旦有了大量数据进行训练，在进行去除语言歧义的测试中几乎有相同的性能（见图 1-20）。

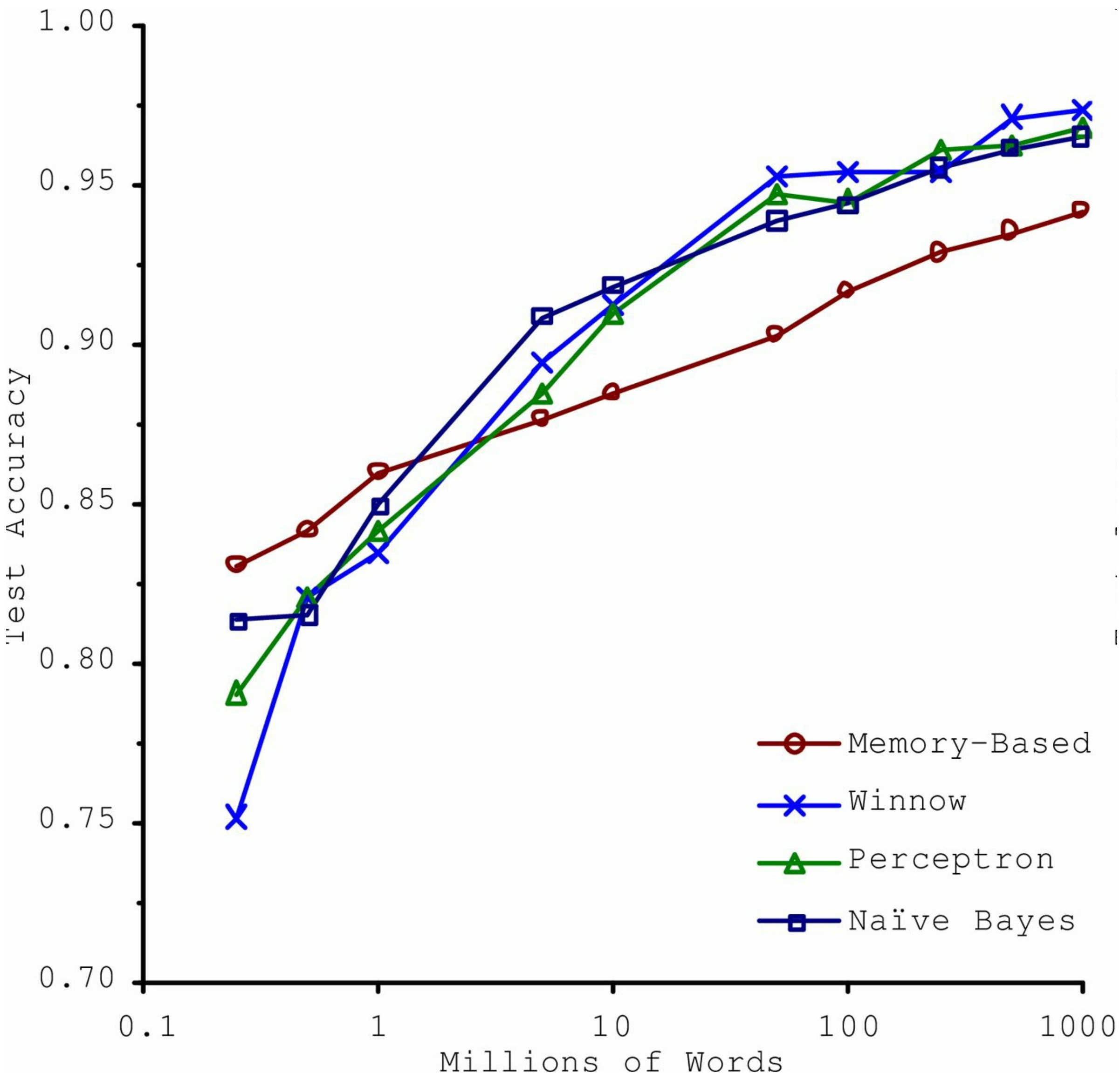


图 1-20 数据和算法的重要性对比

论文作者说："结果说明，我们可能需要重新考虑在算法开发 vs 语料库发展上花费时间和金钱的取舍。"

对于复杂问题，数据比算法更重要的主张在 2009 年由 Norvig 发表的论文《The Unreasonable Effectiveness of Data》得到了进一步的推广。但是，应该注意到，小型和中型的数据集仍然是非常常见的，获得额外的训练数据并不总是轻易和廉价的，所以不要抛弃算法。

没有代表性的训练数据

为了更好地进行归纳推广，让训练数据对新数据具有代表性是非常重要的。无论你用的是基于实例学习或基于模型学习，这点都很重要。

例如，我们之前用来训练线性模型的国家集合不够具有代表性：缺少了一些国家。图 1-21 展示了添加这些缺失国家之后的数据。

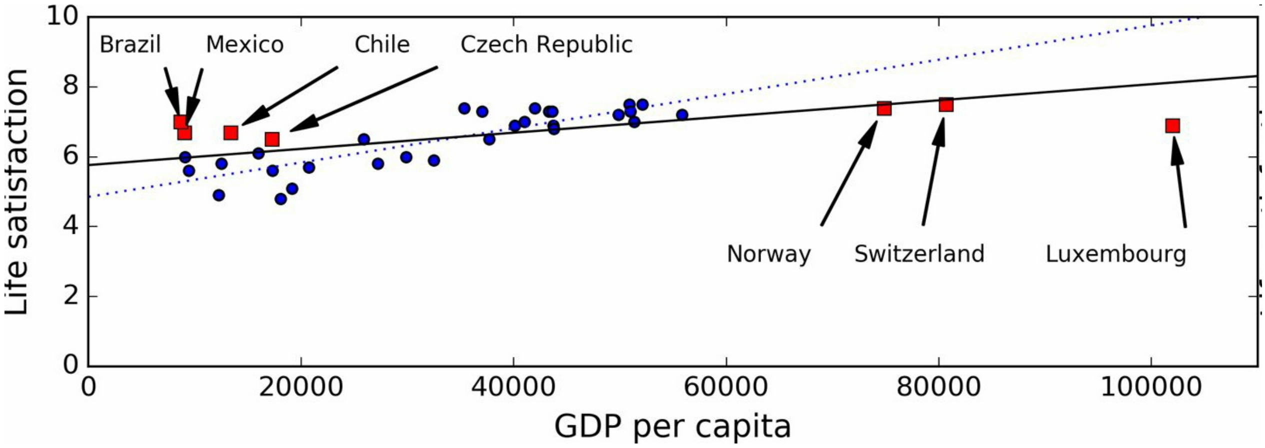


图 1-21 一个更具代表性的训练样本

如果你用这份数据训练线性模型，得到的是实线，旧模型用虚线表示。可以看到，添加几个国家不仅可以显著地改变模型，它还说明如此简单的线性模型可能永远不会达到很好的性能。貌似非常富裕的国家没有中等富裕的国家快乐（事实上，非常富裕的国家看起来更不快乐），相反的，一些贫穷的国家看上去比富裕的国家还幸福。

使用了没有代表性的数据集，我们训练了一个不可能得到准确预测的模型，特别是对于非常贫穷和非常富裕的国家。

使用具有代表性的训练集对于推广到新案例是非常重要的。但是做起来比说起来要难：如果样本太小，就会有样本噪声（即，会有一定概率包含没有代表性的数据），但是即使是非常大的样本也可能没有代表性，如果取样方法错误的话。这叫做样本偏差。

> 一个样本偏差的著名案例
>
> 也许关于样本偏差最有名的案例发生在 1936 年兰登和罗斯福的美国大选：《文学文摘》做了一个非常大的民调，给 1000 万人邮寄了调查信。得到了 240 万回信，非常有信心地预测兰登会以 57% 赢得大选。然而，罗斯福赢得了 62% 的选票。错误发生在《文学文摘》的取样方法：
>
> - 首先，为了获取发信地址，《文学文摘》使用了电话黄页、杂志订阅用户、俱乐部会员等相似的列表。所有这些列表都偏向于富裕人群，他们都倾向于投票给共和党（即兰登）。
> - 第二，只有 25% 的回答了调研。这就又一次引入了样本偏差，它排除了不关心政治的人、不喜欢《文学文摘》的人，和其它关键人群。这种特殊的样本偏差称作无应答偏差。
>
> 下面是另一个例子：假如你想创建一个能识别放克音乐（Funk Music, 别名骚乐）视频的系统。建立训练集的方法之一是在 YouTube 上搜索“放克音乐”，使用搜索到的视频。但是这样就假定了 YouTube 的搜索引擎返回的视频集，是对 YouTube 上的所有放克音乐有代表性的。事实上，搜索结果会偏向于人们歌手（如果你居住在巴西，你会得到许多“funk carioca”视频，它们和 James Brown 的截然不同）。从另一方面来讲，你怎么得到一个大的训练集呢？

低质量数据

很明显，如果训练集中的错误、异常值和噪声（错误测量引入的）太多，系统检测出潜在规律的难度就会变大，性能就会降低。花费时间对训练数据进行清理是十分重要的。事实上，大多数据科学家的一大部分时间是做清洗工作的。例如：

- 如果一些实例是明显的异常值，最好删掉它们或尝试手工修改错误；
- 如果一些实例缺少特征（比如，你的 5% 的顾客没有说明年龄），你必须决定是否忽略这个属性、忽略这些实例、填入缺失值（比如，年龄中位数），或者训练一个含有这个特征的模型和一个不含有这个特征的模型，等等。

不相关的特征

俗语说：进来的是垃圾，出去的也是垃圾。你的系统只有在训练数据包含足够相关特征、非相关特征不多的情况下，才能进行学习。机器学习项目成功的关键之一是用好的特征进行训练。这个过程称作特征工程，包括：

- 特征选择：在所有存在的特征中选取最有用的特征进行训练。
- 特征提取：组合存在的特征，生成一个更有用的特征（如前面看到的，可以使用降维算法）。
- 收集新数据创建新特征。

现在，我们已经看过了许多坏数据的例子，接下来看几个坏算法的例子。

过拟合训练数据

如果你在外国游玩，当地的出租车司机多收了你的钱。你可能会说这个国家所有的出租车司机都是小偷。过度归纳是我们人类经常做的，如果我们不小心，机器也会犯同样的错误。在机器学习中，这称作过拟合：意思是说，模型在训练数据上表现很好，但是推广效果不好。

图 1-22 展示了一个高阶多项式生活满意度模型，它大大过拟合了训练数据。即使它比简单线性模型在训练数据上表现更好，你会相信它的预测吗？

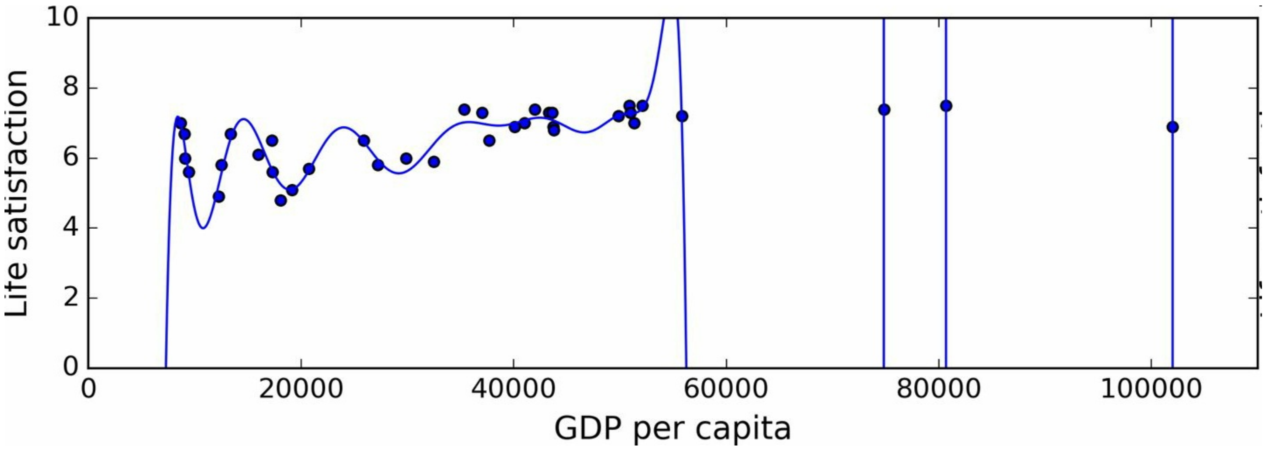


图 1-22 过拟合训练数据

复杂的模型，比如深度神经网络，可以检测数据中的细微规律，但是如果训练集有噪声，或者训练集太小（太小会引入样本噪声），模型就会去检测噪声本身的规律。很明显，这些规律不能推广到新实例。例如，假如你用更多的属性训练生活满意度模型，包括不包含信息的属性，比如国家的名字。如此一来，复杂的模型可能会检测出训练集中名字有 w 字母的国家的生活满意度大于 7：新西兰（7.3），挪威（7.4），瑞典（7.2）和瑞士（7.5）。你能相信这个 W-满意度法则推广到卢旺达和津巴布韦吗？很明显，这个规律只是训练集数据中偶然出现的，但是模型不能判断这个规律是真实的、还是噪声的结果。

> 警告：过拟合发生在相对于训练数据的量和噪声，模型过于复杂的情况。可能的解决方案有：
>
> - 简化模型，可以通过选择一个参数更少的模型（比如使用线性模型，而不是高阶多项式模型）、减少训练数据的属性数、或限制一下模型
> - 收集更多的训练数据
> - 减小训练数据的噪声（比如，修改数据错误和去除异常值）

限定一个模型以让它更简单，降低过拟合的风险被称作正则化（regularization）。例如，我们之前定义的线性模型有两个参数，`θ0` 和 `θ1` 。它给了学习算法两个自由度以让模型适应训练数据：可以调整截距 `θ0` 和斜率 `θ1` 。如果强制 `θ1=0` ，算法就只剩一个自由度，拟合数据就会更为困难：能做的只是将在线下移动，尽可能地靠近训练实例，结果会在平均值附近。

这就是一个非常简单的模型！如果我们允许算法可以修改 `θ1` ，但是只能在一个很小的范围内修改，算法的自由度就会介于 1 和 2 之间。它要比两个自由度的模型简单，比 1 个自由度的模型要复杂。你的目标是在完美拟合数据和保持模型简单性上找到平衡，确保算法的推广效果。

图 1-23 展示了三个模型：虚线表示用缺失部分国家的数据训练的原始模型，短划线是我们的第二个用所有国家训练的模型，实线模型的训练数据和第一个相同，但进行了正则化限制。你可以看到正则化强制模型有一个小的斜率，它对训练数据的拟合不是那么好，但是对新样本的推广效果好。

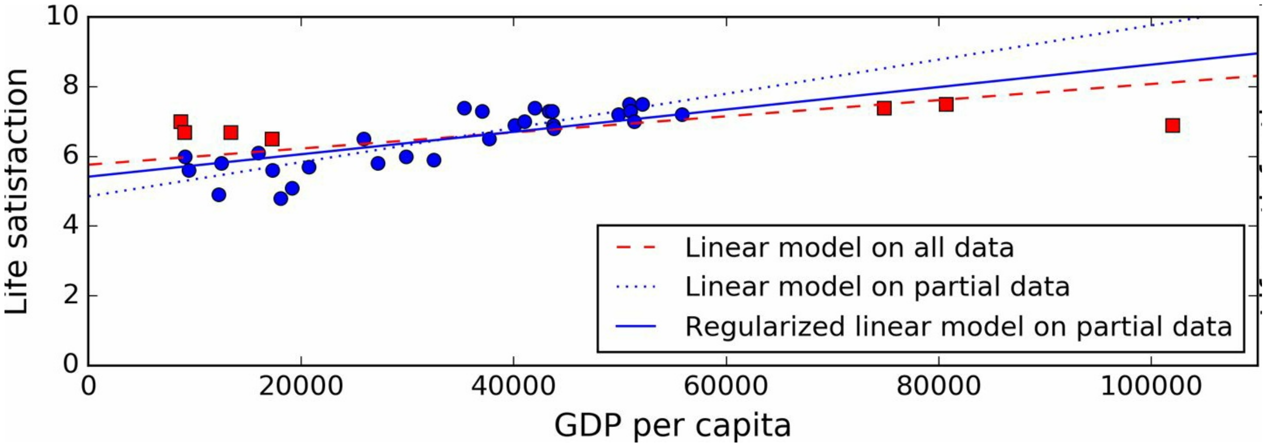


图 1-23 正则化降低了过度拟合的风险

正则化的度可以用一个超参数（hyperparameter）控制。超参数是一个学习算法的参数（而不是模型的）。这样，它是不会被学习算法本身影响的，它优于训练，在训练中是保持不变的。如果你设定的超参数非常大，就会得到一个几乎是平的模型（斜率接近于 0）；这种学习算法几乎肯定不会过拟合训练数据，但是也很难得到一个好的解。调节超参数是创建机器学习算法非常重要的一部分（下一章你会看到一个详细的例子）。

欠拟合训练数据

你可能猜到了，欠拟合是和过拟合相对的：当你的模型过于简单时就会发生。例如，生活满意度的线性模型倾向于欠拟合；现实要比这个模型复杂的多，所以预测很难准确，即使在训练样本上也很难准确。

解决这个问题的选项包括：

- 选择一个更强大的模型，带有更多参数
- 用更好的特征训练学习算法（特征工程）
- 减小对模型的限制（比如，减小正则化超参数）

回顾

现在，你已经知道了很多关于机器学习的知识。然而，学过了这么多概念，你可能会感到有些迷失，所以让我们退回去，回顾一下重要的：

- 机器学习是让机器通过学习数据对某些任务做得更好，而不使用确定的代码规则。
- 有许多不同类型的机器学习系统：监督或非监督，批量或在线，基于实例或基于模型，等等。
- 在机器学习项目中，我们从训练集中收集数据，然后对学习算法进行训练。如果算法是基于模型的，就调节一些参数，让模型拟合到训练集（即，对训练集本身作出好的预测），然后希望它对新样本也能有好预测。如果算法是基于实例的，就是用记忆学习样本，然后用相似度推广到新实例。
- 如果训练集太小、数据没有代表性、含有噪声、或掺有不相关的特征（垃圾进，垃圾出），系统的性能不会好。最后，模型不能太简单（会发生欠拟合）或太复杂（会发生过拟合）。

还差最后一个主题要学习：训练完了一个模型，你不只希望将它推广到新样本。如果你想评估它，那么还需要作出必要的微调。一起来看一看。

测试和确认

要知道一个模型推广到新样本的效果，唯一的办法就是真正的进行试验。一种方法是将模型部署到生产环境，观察它的性能。这么做可以，但是如果模型的性能很差，就会引起用户抱怨 —— 这不是最好的方法。

更好的选项是将你的数据分成两个集合：训练集和测试集。正如它们的名字，用训练集进行训练，用测试集进行测试。对新样本的错误率称作推广错误（或样本外错误），通过模型对测试集的评估，你可以预估这个错误。这个值可以告诉你，你的模型对新样本的性能。

如果训练错误率低（即，你的模型在训练集上错误不多），但是推广错误率高，意味着模型对训练数据过拟合。

> 提示：一般使用 80% 的数据进行训练，保留20%用于测试。

因此，评估一个模型很简单：只要使用测试集。现在假设你在两个模型之间犹豫不决（比如一个线性模型和一个多项式模型）：如何做决定呢？一种方法是两个都训练，，然后比较在测试集上的效果。

现在假设线性模型的效果更好，但是你想做一些正则化以避免过拟合。问题是：如何选择正则化超参数的值？一种选项是用 100 个不同的超参数训练100个不同的模型。假设你发现最佳的超参数的推广错误率最低，比如只有 5%。然后就选用这个模型作为生产环境，但是实际中性能不佳，误差率达到了 15%。发生了什么呢？

答案在于，你在测试集上多次测量了推广误差率，调整了模型和超参数，以使模型最适合这个集合。这意味着模型对新数据的性能不会高。

这个问题通常的解决方案是，再保留一个集合，称作验证集合。用训练集和多个超参数训练多个模型，选择在验证集上有最佳性能的模型和超参数。当你对模型满意时，用测试集再做最后一次测试，以得到推广误差率的预估。

为了避免“浪费”过多训练数据在验证集上，通常的办法是使用交叉验证：训练集分成互补的子集，每个模型用不同的子集训练，再用剩下的子集验证。一旦确定模型类型和超参数，最终的模型使用这些超参数和全部的训练集进行训练，用测试集得到推广误差率。

> 没有免费午餐公理
>
> 模型是观察的简化版本。简化意味着舍弃无法进行推广的表面细节。但是，要确定舍弃什么数据、保留什么数据，必须要做假设。例如，线性模型的假设是数据基本上是线性的，实例和模型直线间的距离只是噪音，可以放心忽略。
>
> 在一篇 1996 年的著名论文中，David Wolpert 证明，如果完全不对数据做假设，就没有理由选择一个模型而不选另一个。这称作没有免费午餐（NFL）公理。对于一些数据集，最佳模型是线性模型，而对其它数据集是神经网络。没有一个模型可以保证效果更好（如这个公理的名字所示）。确信的唯一方法就是测试所有的模型。因为这是不可能的，实际中就必须要做一些对数据合理的假设，只评估几个合理的模型。例如，对于简单任务，你可能是用不同程度的正则化评估线性模型，对于复杂问题，你可能要评估几个神经网络模型。

练习

本章中，我们学习了一些机器学习中最为重要的概念。下一章，我们会更加深入，并写一些代码。开始下章之前，确保你能回答下面的问题：

1. 如何定义机器学习？
2. 机器学习可以解决的四类问题？
3. 什么是带标签的训练集？
4. 最常见的两个监督任务是什么？
5. 指出四个常见的非监督任务？
6. 要让一个机器人能在各种未知地形行走，你会采用什么机器学习算法？
7. 要对你的顾客进行分组，你会采用哪类算法？
8. 垃圾邮件检测是监督学习问题，还是非监督学习问题？
9. 什么是在线学习系统？
10. 什么是核外学习？
11. 什么学习算法是用相似度做预测？
12. 模型参数和学习算法的超参数的区别是什么？
13. 基于模型学习的算法搜寻的是什么？最成功的策略是什么？基于模型学习如何做预测？
14. 机器学习的四个主要挑战是什么？
15. 如果模型在训练集上表现好，但推广到新实例表现差，问题是什么？给出三个可能的解

决方案。

16. 什么是测试集，为什么要使用它？
17. 验证集的目的是什么？
18. 如果用测试集调节超参数，会发生什么？
19. 什么是交叉验证，为什么它比验证集好？

练习答案见附录A。

第2章 一个完整的机器学习项目

本章中，你会假装作为被一家地产公司刚刚雇佣的数据科学家，完整地学习一个案例项目。下面是主要步骤：

1. 项目概述。
2. 获取数据。
3. 发现并可视化数据，发现规律。
4. 为机器学习算法准备数据。
5. 选择模型，进行训练。
6. 微调模型。
7. 给出解决方案。
8. 部署、监控、维护系统。

使用真实数据

学习机器学习时，最好使用真实数据，而不是人工数据集。幸运的是，有上千个开源数据集可以进行选择，涵盖多个领域。以下是一些可以查找的数据的地方：

- 流行的开源数据仓库：
  - UC Irvine Machine Learning Repository
  - Kaggle datasets
  - Amazon's AWS datasets
- 准入口（提供开源数据列表）
  - http://dataportals.org/
  - http://opendatamonitor.eu/
  - http://quandl.com/
- 其它列出流行开源数据仓库的网页：
  - Wikipedia's list of Machine Learning datasets
  - Quora.com question
  - Datasets subreddit

本章，我们选择的是 StatLib 的加州房产价格数据集（见图 2-1）。这个数据集是基于 1990 年加州普查的数据。数据已经有点老（1990 年还能买一个湾区不错的房子），但是它有许多优点，利于学习，所以假设这个数据为最近的。为了便于教学，我们添加了一个类别属性，并除去了一些。

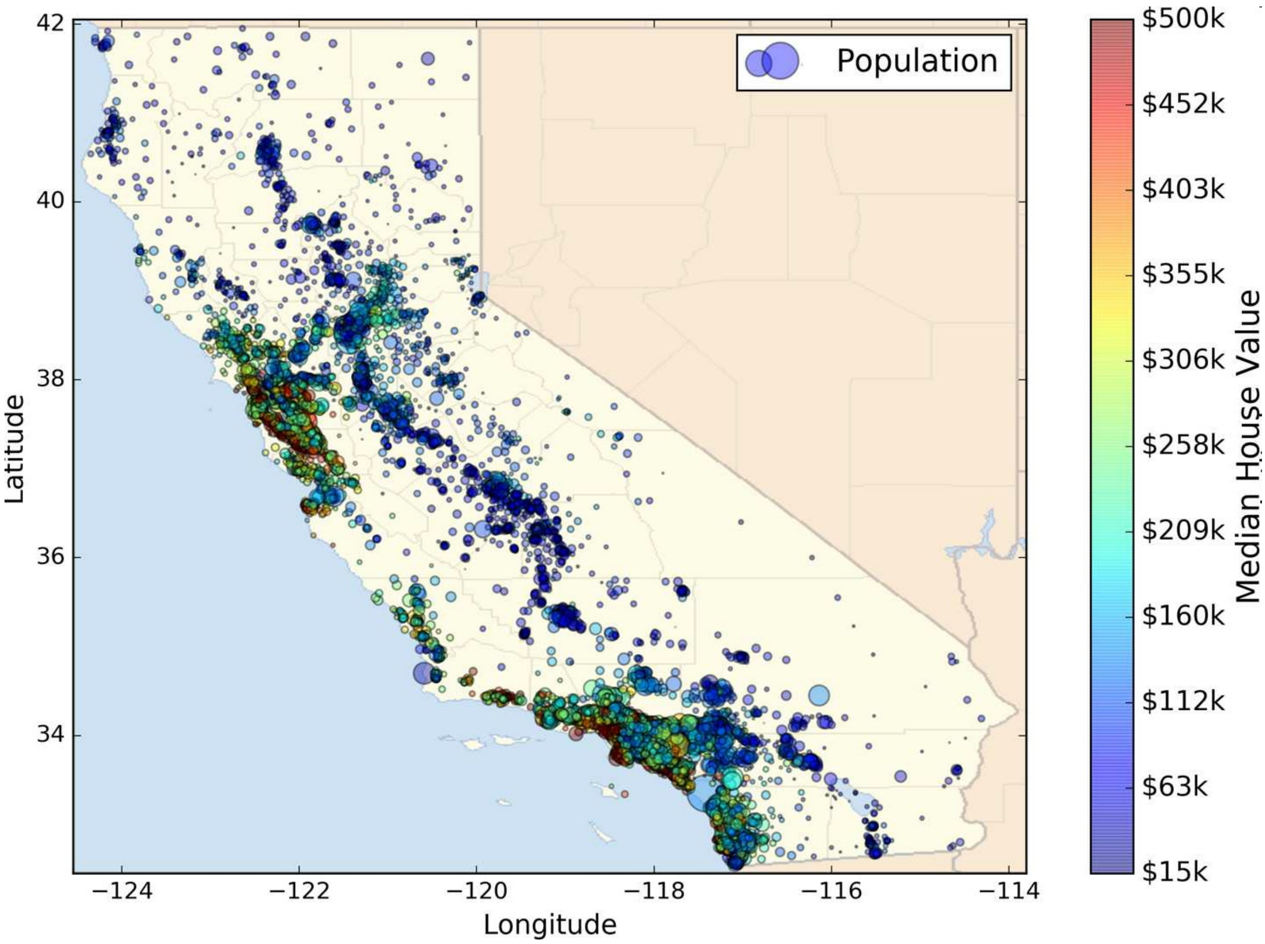


图 2-1 加州房产价格

项目概览

欢迎来到机器学习房地产公司！你的第一个任务是利用加州普查数据，建立一个加州房价模型。这个数据包含每个街区组的人口、收入中位数、房价中位数等指标。

街区组是美国调查局发布样本数据的最小地理单位（一个街区通常有 600 到 3000 人）。我们将其简称为“街区”。

你的模型要利用这个数据进行学习，然后根据其它指标，预测任何街区的的房价中位数。

> 提示：你是一个有条理的数据科学家，你要做的第一件事是拿出你的机器学习项目清单。你可以使用附录 B 中的清单；这个清单适用于大多数的机器学习项目，但是你还是要确认它是否满足需求。在本章中，我们会检查许多清单上的项目，但是也会跳过一些简单的，有些会在后面的章节再讨论。

划定问题

问老板的第一个问题应该是商业目标是什么？建立模型可能不是最终目标。公司要如何使用、并从模型受益？这非常重要，因为它决定了如何划定问题，要选择什么算法，评估模型性能的指标是什么，要花多少精力进行微调。

老板告诉你你的模型的输出（一个区的房价中位数）会传给另一个机器学习系统（见图 2-2），也有其它信号会传入后面的系统。这一整套系统可以确定某个区进行投资值不值。确定值不值得投资非常重要，它直接影响利润。

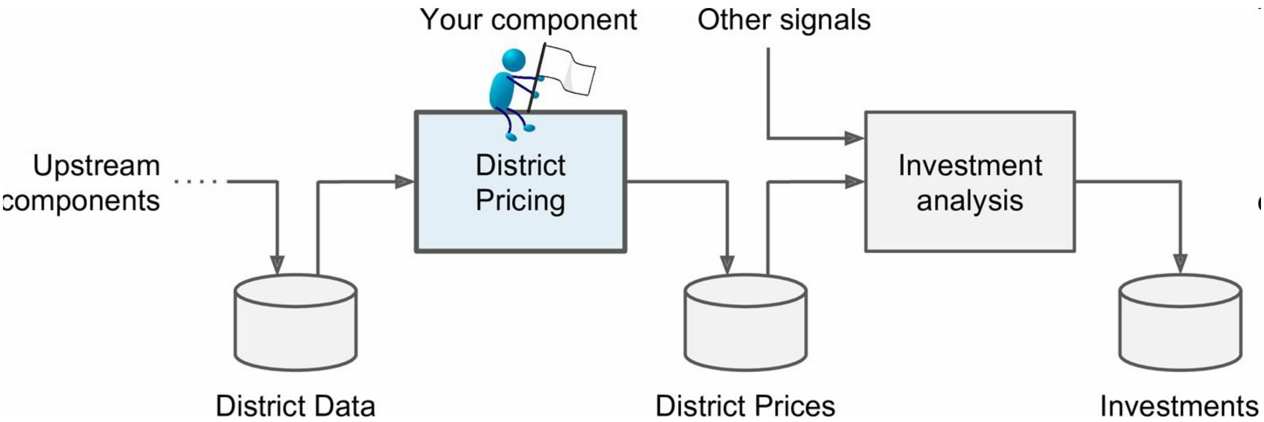


图 2-2 房地产投资的机器学习流水线

> 流水线
>
> 一系列的数据处理组件被称为数据流水线。流水线在机器学习系统中很常见，因为有许多数据要处理和转换。
>
> 组件通常是异步运行的。每个组件吸纳进大量数据，进行处理，然后将数据传输到另一个数据容器中，而后流水线中的另一个组件收入这个数据，然后输出，这个过程依次进行下去。每个组件都是独立的：组件间的接口只是数据容器。这样可以让系统更便于理解（记住数据流的图），不同的项目组可以关注于不同的组件。进而，如果一个组件失效了，下游的组件使用失效组件最后生产的数据，通常可以正常运行（一段时间）。这样就使整个架构相当健壮。
>
> 另一方面，如果没有监控，失效的组件会在不被注意的情况下运行一段时间。数据会受到污染，整个系统的性能就会下降。

下一个要问的问题是，现在的解决方案效果如何。老板通常会给一个参考性能，以及如何解决问题。老板说，现在街区的房价是靠专家手工估计的，专家队伍收集最新的关于一个区的信息（不包括房价中位数），他们使用复杂的规则进行估计。这种方法费钱费时间，而且估计结果不理想，误差率大概有 15%。

OK，有了这些信息，你就可以开始设计系统了。首先，你需要划定问题：监督或非监督，还是强化学习？这是个分类任务、回归任务，还是其它的？要使用批量学习还是线上学习？继续阅读之前，请暂停一下，尝试自己回答下这些问题。

你能回答出来吗？一起看下答案：很明显，这是一个典型的监督学习任务，因为你要使用的是有标签的训练样本（每个实例都有预定的产出，即街区的房价中位数）。并且，这是一个典型的回归任务，因为你要预测一个值。讲的更细些，这是一个多变量回归问题，因为系统要使用多个变量进行预测（要使用街区的人口，收入中位数等等）。在第一章中，你只是根据人均 GDP 来预测生活满意度，因此这是一个单变量回归问题。最后，没有连续的数据流进入系统，没有特别需求需要对数据变动作出快速适应。数据量不大可以放到内存中，因此批量学习就够了。

> 提示：如果数据量很大，你可以要么在多个服务器上对批量学习做拆分（使用 MapReduce 技术，后面会看到），或是使用线上学习。

选择性能指标

下一步是选择性能指标。回归问题的典型指标是均方根误差（RMSE）。均方根误差测量的是系统预测误差的标准差。例如，RMSE 等于 50000，意味着，68% 的系统预测值位于实际值的 50000 美元以内，95% 的预测值位于实际值的 100000 美元以内（一个特征通常都符合高斯分布，即满足 “68-95-99.7”规则：大约68%的值落在`1σ`内，95% 的值落在`2σ`内，99.7% 的值落在`3σ`内，这里的`σ`等于50000）。公式 2-1 展示了计算 RMSE 的方法。

$$\mathrm{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(h(\mathbf{x}^{(i)}) - y^{(i)}\right)^2}$$

公式 2-1 均方根误差（RMSE）

> 符号的含义
>
> 这个方程引入了一些常见的贯穿本书的机器学习符号：
>
> - `m`是测量 RMSE 的数据集中的实例数量。
>   例如，如果用一个含有 2000 个街区的验证集求 RMSE，则`m = 2000`。
>
> - $x^{(i)}$ 是数据集第`i`个实例的所有特征值（不包含标签）的向量，$y^{(i)}$ 是它的标签（这个实例的输出值）。
>
>   例如，如果数据集中的第一个街区位于经度 –118.29°，纬度 33.91°，有 1416 名居民，收入中位数是 38372 美元，房价中位数是 156400 美元（忽略掉其它的特征），则有：

$$\mathbf{x}^{(1)} = \begin{pmatrix} -118.29 \\ 33.91 \\ 1,416 \\ 38,372 \end{pmatrix}$$

和，

$$y^{(1)} = 156,400$$

- `X` 是包含数据集中所有实例的所有特征值（不包含标签）的矩阵。每一行是一个实例，第 `i` 行是 $x^{(i)}$ 的转置，记为 $x^{(i)T}$ 。

  例如，仍然是前面提到的第一区，矩阵 `X` 就是：

$$\mathbf{X} = \begin{pmatrix} (\mathbf{x}^{(1)})^T \\ (\mathbf{x}^{(2)})^T \\ \vdots \\ (\mathbf{x}^{(1999)})^T \\ (\mathbf{x}^{(2000)})^T \end{pmatrix} = \begin{pmatrix} -118.29 & 33.91 & 1,416 & 38,372 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

- `h` 是系统的预测函数，也称为假设（hypothesis）。当系统收到一个实例的特征向量 $x^{(i)}$，就会输出这个实例的一个预测值 $\hat{y}^{(i)} = h(x^{(i)})$（$\hat{y}$ 读作 `y-hat` ）。

  例如，如果系统预测第一区的房价中位数是 158400 美元，则 $\hat{y}^{(1)} = h(x^{(1)}) = 158400$ 。预测误差是 $\hat{y}^{(1)} y^{(1)} = 2000$ 。

- `RMSE(X,h)` 是使用假设 `h` 在样本集上测量的损失函数。

我们使用小写斜体表示标量值（例如 m 或 $y^{(i)}$）和函数名（例如 h），小写粗体表示向量（例如 $x^{(i)}$），大写粗体表示矩阵（例如 X）。

虽然大多数时候 RMSE 是回归任务可靠的性能指标，在有些情况下，你可能需要另外的函数。例如，假设存在许多异常的街区。此时，你可能需要使用平均绝对误差（Mean Absolute Error，也称作平均绝对偏差），见公式 2-2：

$$\mathrm{MAE}(\mathbf{X}, h) = \frac{1}{m}\sum_{i=1}^{m} | h(\mathbf{x}^{(i)}) - y^{(i)} |$$

公式2-2 平均绝对误差

RMSE 和 MAE 都是测量预测值和目标值两个向量距离的方法。有多种测量距离的方法，或范数：

- 计算对应欧几里得范数的平方和的根（RMSE）：这个距离介绍过。它也称作 `ℓ2` 范数，标记为 $\|\cdot\|_2$（或只是 $\|\cdot\|$）。
- 计算对应于 `ℓ1` （标记为 $\|\cdot\|_1$）范数的绝对值和（MAE）。有时，也称其为曼哈顿范数，因为它测量了城市中的两点，沿着矩形的边行走的距离。
- 更一般的，包含 `n` 个元素的向量 `v` 的 `ℓk` 范数（K 阶闵氏范数），定义成

  $$\| \mathbf{v} \|_k = (| v_0 |^k + | v_1 |^k + \cdots + | v_n |^k)^{\frac{1}{k}}$$

  `ℓ0` （汉明范数）只显示了这个向量的基数（即，非零元素的个数），`ℓ∞` （切比雪夫范数）是向量中最大的绝对值。
- 范数的指数越高，就越关注大的值而忽略小的值。这就是为什么 RMSE 比 MAE 对异常值更敏感。但是当异常值是指数分布的（类似正态曲线），RMSE 就会表现很好。

核实假设

最后，最好列出并核对迄今（你或其他人）作出的假设，这样可以尽早发现严重的问题。例如，你的系统输出的街区房价，会传入到下游的机器学习系统，我们假设这些价格确实会被当做街区房价使用。但是如果下游系统实际上将价格转化成了分类（例如，便宜、中等、昂贵），然后使用这些分类，而不是使用价格。这样的话，获得准确的价格就不那么重要了，你只需要得到合适的分类。问题相应地就变成了一个分类问题，而不是回归任务。你可不想在一个回归系统上工作了数月，最后才发现真相。

幸运的是，在与下游系统主管探讨之后，你很确信他们需要的就是实际的价格，而不是分类。很好！整装待发，可以开始写代码了。

获取数据

开始动手。最后用 Jupyter notebook 完整地敲一遍示例代码。完整的代码位于 https://github.com/ageron/handson-ml。

创建工作空间

首先，你需要安装 Python。可能已经安装过了，没有的话，可以从官网下载 https://www.python.org/。

接下来，需要为你的机器学习代码和数据集创建工作空间目录。打开一个终端，输入以下命令（在提示符 `$` 之后）：

```
$ export ML_PATH="$HOME/ml"      # 可以更改路径
$ mkdir -p $ML_PATH
```

还需要一些 Python 模块：Jupyter、NumPy、Pandas、Matplotlib 和 Scikit-Learn。如果所有这些模块都已经在 Jupyter 中运行了，你可以直接跳到下一节“下载数据”。如果还没安装，有多种方法可以进行安装（包括它们的依赖）。你可以使用系统的包管理系统（比如 Ubuntu 上的 `apt-get`，或 macOS 上的 MacPorts 或 HomeBrew），安装一个 Python 科学计算环境比如 Anaconda，使用 Anaconda 的包管理系统，或者使用 Python 自己的包管理器 `pip`，它是 Python 安装包（自从 2.7.9 版本）自带的。可以用下面的命令检测是否安装 `pip`：

```
$ pip3 --version
pip 9.0.1 from [...]/lib/python3.5/site-packages (python 3.5)
```

你需要保证 `pip` 是近期的版本，至少高于 1.4，以保障二进制模块文件的安装（也称为 wheel）。要升级 `pip`，可以使用下面的命令：

```
$ pip3 install --upgrade pip
Collecting pip
[...]
Successfully installed pip-9.0.1
```

> 创建独立环境
>
> 如果你希望在一个独立环境中工作（强烈推荐这么做，不同项目的库的版本不会冲突），用下面的 `pip` 命令安装 `virtualenv`：
>
> ```
> $ pip3 install --user --upgrade virtualenv
> Collecting virtualenv
> [...]
> Successfully installed virtualenv
> ```
>
> 现在可以通过下面命令创建一个独立的 Python 环境：
>
> ```
> $ cd $ML_PATH
> $ virtualenv env
> Using base prefix '[...]'
> New python executable in [...]/ml/env/bin/python3.5
> Also creating executable in [...]/ml/env/bin/python
> Installing setuptools, pip, wheel...done.
> ```
>
> 以后每次想要激活这个环境,只需打开一个终端然后输入：
>
> ```
> $ cd $ML_PATH
> $ source env/bin/activate
> ```
>
> 启动该环境时，使用 `pip` 安装的任何包都只安装于这个独立环境中，Python 指挥访问这些包（如果你希望 Python 能访问系统的包，创建环境时要使用包选项 `--system-site`）。更多信息，请查看 `virtualenv` 文档。

现在，你可以使用 `pip` 命令安装所有必需的模块和它们的依赖：

```
$ pip3 install --upgrade jupyter matplotlib numpy pandas scipy scikit-learn
Collecting jupyter
  Downloading jupyter-1.0.0-py2.py3-none-any.whl
Collecting matplotlib
  [...]
```

要检查安装，可以用下面的命令引入每个模块：

```
$ python3 -c "import jupyter, matplotlib, numpy, pandas, scipy, sklearn"
```

这个命令不应该有任何输出和错误。现在你可以用下面的命令打开 Jupyter：

```
$ jupyter notebook
[I 15:24 NotebookApp] Serving notebooks from local directory: [...]/ml
[I 15:24 NotebookApp] 0 active kernels
[I 15:24 NotebookApp] The Jupyter Notebook is running at: http://localhost:8888/
[I 15:24 NotebookApp] Use Control-C to stop this server and shut down all
kernels (twice to skip confirmation).
```

Jupyter 服务器现在运行在终端上，监听 888 8端口。你可以用浏览器打开 `http://localhost:8888/` ，以访问这个服务器（服务器启动时，通常就自动打开了）。你可以看到一个空的工作空间目录（如果按照先前的 `virtualenv` 步骤，只包含 `env` 目录）。

现在点击按钮 New 创建一个新的 Python 注本，选择合适的 Python 版本（见图 2-3）。

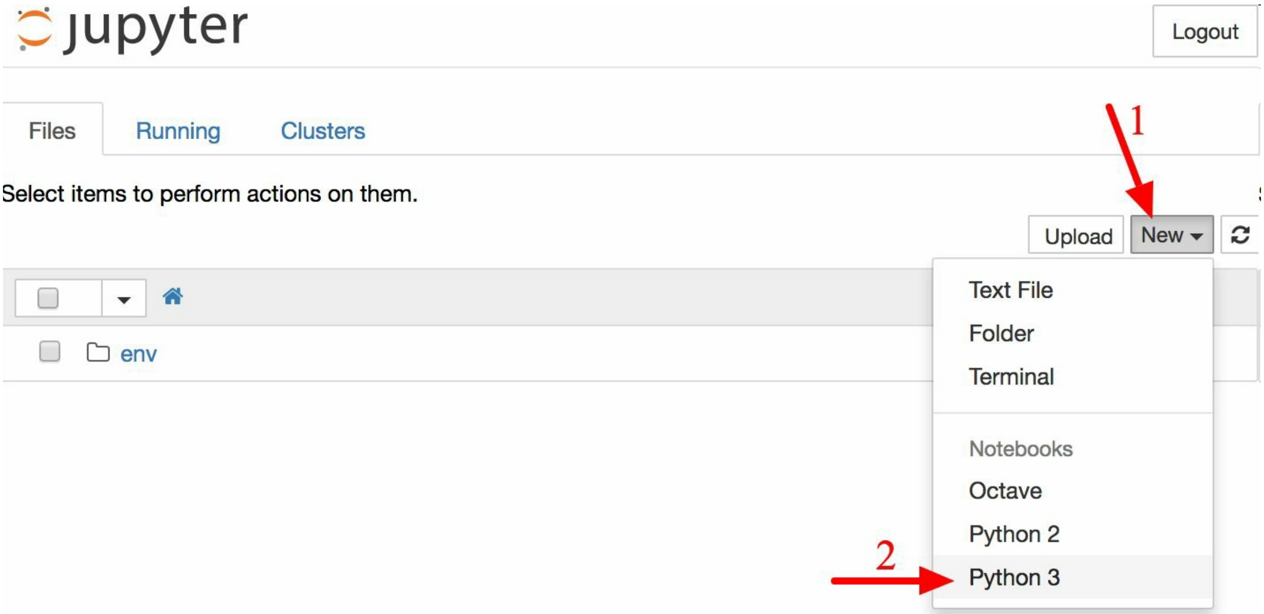


图 2-3 Jupyter 的工作空间

这一步做了三件事：首先，在工作空间中创建了一个新的 notebook 文件 `Untitled.ipynb` ；第二，它启动了一个 Jupyter 的 Python 内核来运行这个 notebook；第三，在一个新栏中打开这个 notebook。接下来，点击 Untitled，将这个 notebook 重命名为 `Housing` （这会将 `ipynb` 文件自动命名为 `Housing.ipynb` ）。

notebook 包含一组代码框。每个代码框可以放入可执行代码或格式化文本。现在，notebook 只有一个空的代码框，标签是 `In [1]:` 。在框中输入 `print("Hello world!")` ，点击运行按钮（见图 2-4）或按 `Shift+Enter` 。这会将当前的代码框发送到 Python 内核，运行之后会返回输出。结果显示在代码框下方。由于抵达了 notebook 的底部，一个新的代码框会被自动创建出来。从 Jupyter 的 Help 菜单中的 User Interface Tour，可以学习 Jupyter 的基本操作。

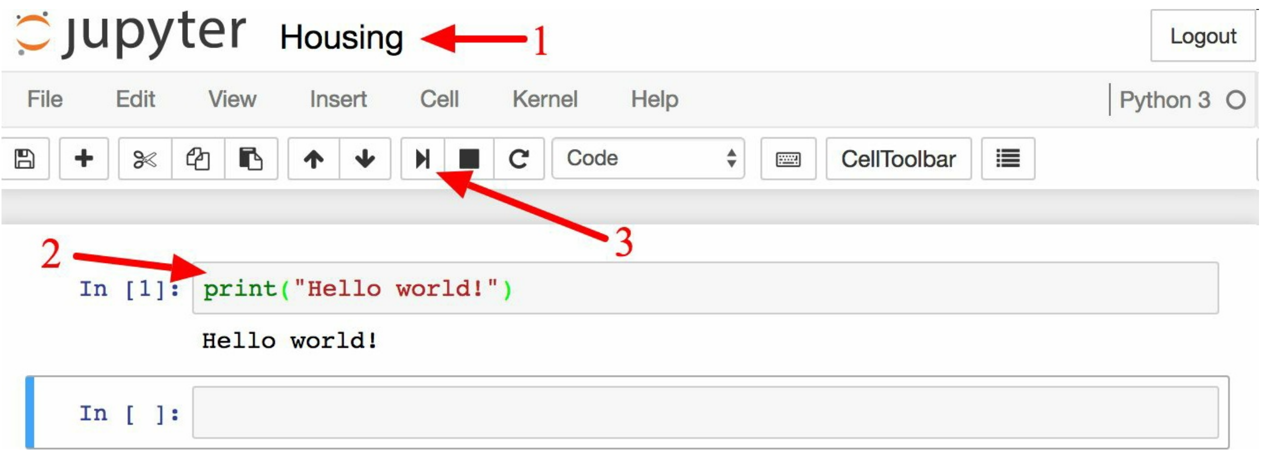


图 2-4 在 notebook 中打印`Hello world!`

下载数据

一般情况下，数据是存储于关系型数据库（或其它常见数据库）中的多个表、文档、文件。要访问数据，你首先要有密码和登录权限，并要了解数据模式。但是在这个项目中，这一切要简单些：只要下载一个压缩文件，`housing.tgz`，它包含一个 CSV 文件`housing.csv`，含有所有数据。

你可以使用浏览器下载，运行`tar xzf housing.tgz`解压出`csv`文件，但是更好的办法是写一个小函数来做这件事。如果数据变动频繁，这么做是非常好的，因为可以让你写一个小脚本随时获取最新的数据（或者创建一个定时任务来做）。如果你想在多台机器上安装数据集，获取数据自动化也是非常好的。

下面是获取数据的函数：

```
import os
import tarfile
from six.moves import urllib

DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml/master/"
HOUSING_PATH = "datasets/housing"
HOUSING_URL = DOWNLOAD_ROOT + HOUSING_PATH + "/housing.tgz"

def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
    if not os.path.isdir(housing_path):
        os.makedirs(housing_path)
    tgz_path = os.path.join(housing_path, "housing.tgz")
    urllib.request.urlretrieve(housing_url, tgz_path)
    housing_tgz = tarfile.open(tgz_path)
    housing_tgz.extractall(path=housing_path)
    housing_tgz.close()
```

现在，当你调用`fetch_housing_data()`，就会在工作空间创建一个`datasets/housing`目录，下载`housing.tgz`文件，解压出`housing.csv`。

然后使用Pandas加载数据。还是用一个小函数来加载数据：

```
import pandas as pd

def load_housing_data(housing_path=HOUSING_PATH):
    csv_path = os.path.join(housing_path, "housing.csv")
    return pd.read_csv(csv_path)
```

这个函数会返回一个包含所有数据的 Pandas `DataFrame` 对象。

快速查看数据结构

使用`DataFrame`的`head()`方法查看该数据集的前5行（见图 2-5）。

```
In [5]: housing = load_housing_data()
        housing.head()
```

Out[5]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	populatio
0	-122.23	37.88	41.0	880.0	129.0	322.0
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0
2	-122.24	37.85	52.0	1467.0	190.0	496.0
3	-122.25	37.85	52.0	1274.0	235.0	558.0
4	-122.25	37.85	52.0	1627.0	280.0	565.0

图 2-5 数据集的前五行

每一行都表示一个街区。共有 10 个属性（截图中可以看到 6 个）：经度、维度、房屋年龄中位数、总房间数、总卧室数、人口数、家庭数、收入中位数、房屋价值中位数、离大海距离。

`info()` 方法可以快速查看数据的描述，特别是总行数、每个属性的类型和非空值的数量（见图 2-6）。

```
In [6]: housing.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
longitude             20640 non-null float64
latitude              20640 non-null float64
housing_median_age    20640 non-null float64
total_rooms           20640 non-null float64
total_bedrooms        20433 non-null float64
population            20640 non-null float64
households            20640 non-null float64
median_income         20640 non-null float64
median_house_value    20640 non-null float64
ocean_proximity       20640 non-null object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

图 2-6 房屋信息

数据集中共有 20640 个实例，按照机器学习的标准这个数据量很小，但是非常适合入门。我们注意到总房间数只有 20433 个非空值，这意味着有 207 个街区缺少这个值。我们将在后面对它进行处理。

所有的属性都是数值的，除了离大海距离这项。它的类型是对象，因此可以包含任意 Python 对象，但是因为该项是从 CSV 文件加载的，所以必然是文本类型。在刚才查看数据前五项时，你可能注意到那一列的值是重复的，意味着它可能是一项表示类别的属性。可以使用 `value_counts()` 方法查看该项中都有哪些类别，每个类别中都包含有多少个街区：

```
>>> housing["ocean_proximity"].value_counts()
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
Name: ocean_proximity, dtype: int64
```

再来看其它字段。`describe()` 方法展示了数值属性的概括（见图 2-7）。

```
In [8]: housing.describe()
```

Out[8]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrc
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000C
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553
std	2.003532	2.135952	12.585558	2181.615252	421.38507C
min	-124.350000	32.540000	1.000000	2.000000	1.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.00000C
50%	-118.490000	34.260000	29.000000	2127.000000	435.00000C
75%	-118.010000	37.710000	37.000000	3148.000000	647.00000C
max	-114.310000	41.950000	52.000000	39320.000000	6445.0000C

图 2-7 每个数值属性的概括

`count`、`mean`、`min` 和 `max` 几行的意思很明显了。注意，空值被忽略了（所以，卧室总数是 20433 而不是 20640）。`std` 是标准差（揭示数值的分散度）。25%、50%、75% 展示了对应的分位数：每个分位数指明小于这个值，且指定分组的百分比。例如，25% 的街区的房屋年龄中位数小于 18，而 50% 的小于 29，75% 的小于 37。这些值通常称为第 25 个百分位数（或第一个四分位数），中位数，第 75 个百分位数（第三个四分位数）。

另一种快速了解数据类型的方法是画出每个数值属性的柱状图。柱状图（的纵轴）展示了特定范围的实例的个数。你还可以一次给一个属性画图，或对完整数据集调用 `hist()` 方法，后者会画出每个数值属性的柱状图（见图 2-8）。例如，你可以看到略微超过 800 个街区的 `median_house_value` 值差不多等于 500000 美元。

```
%matplotlib inline   # only in a Jupyter notebook
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20,15))
plt.show()
```

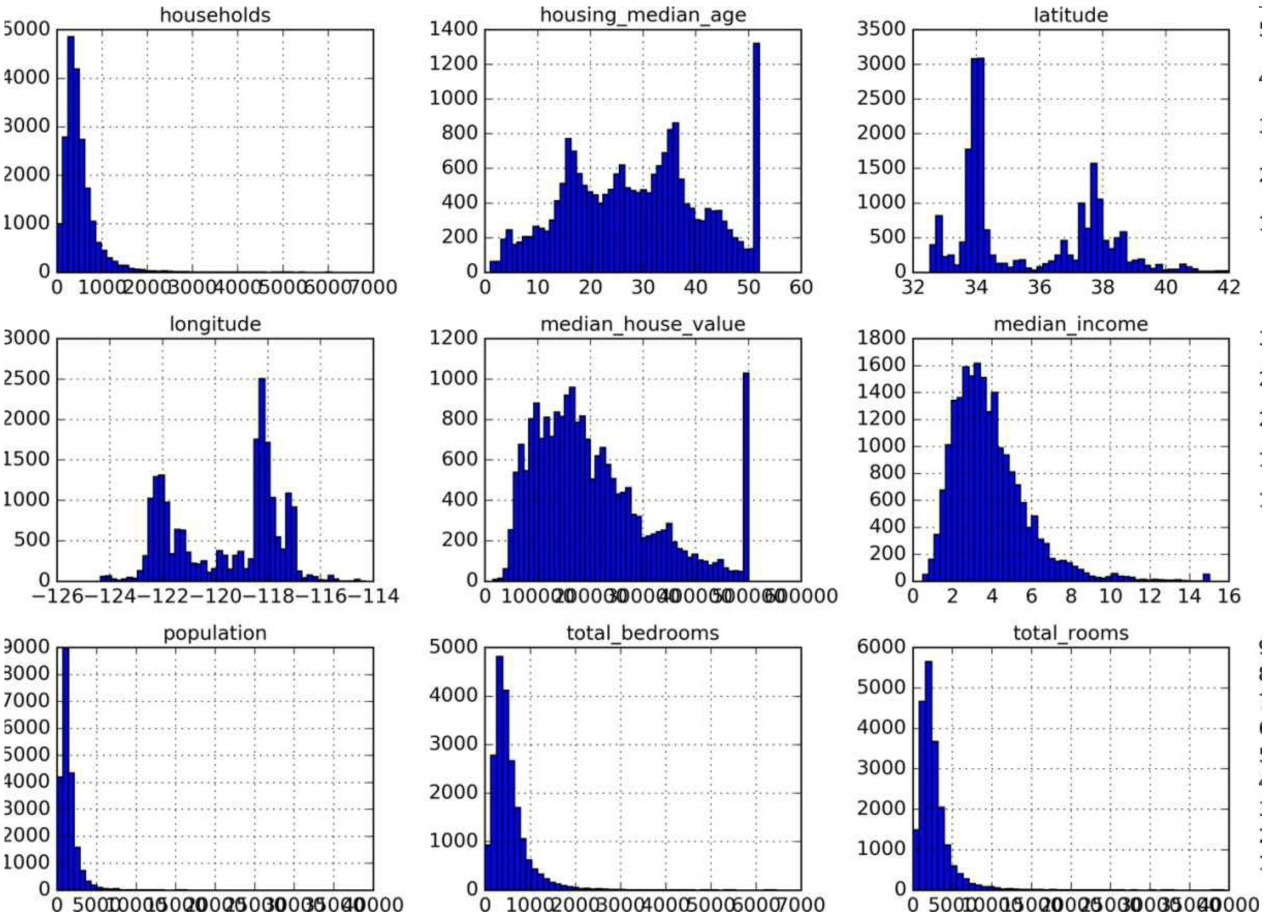


图 2-8 每个数值属性的柱状图

> 注：`hist()` 方法依赖于 Matplotlib，后者依赖于用户指定的图形后端以打印到屏幕上。因此在画图之前，你要指定 Matplotlib 要使用的后端。最简单的方法是使用 Jupyter 的魔术命令 `%matplotlib inline`。它会告诉 Jupyter 设定好 Matplotlib，以使用 Jupyter 自己的后端。绘图就会在 notebook 中渲染了。注意在 Jupyter 中调用 `show()` 不是必要的，因为代码框执行后 Jupyter 会自动展示图像。

注意柱状图中的一些点：

1. 首先，收入中位数貌似不是美元（USD）。与数据采集团队交流之后，你被告知数据是经过缩放调整的，过高收入中位数的会变为 15（实际为 15.0001），过低的会变为 5（实际为 0.4999）。在机器学习中对数据进行预处理很正常，这不一定是个问题，但你要明白数据是如何计算出来的。

2. 房屋年龄中位数和房屋价值中位数也被设了上限。后者可能是个严重的问题，因为它是你的目标属性（你的标签）。你的机器学习算法可能学习到价格不会超出这个界限。你需要与下游团队核实，这是否会成为问题。如果他们告诉你他们需要明确的预测值，即使超过 500000 美元，你则有两个选项：

    i. 对于设了上限的标签，重新收集合适的标签；
    ii. 将这些街区从训练集移除（也从测试集移除，因为若房价超出 500000 美元，你的系统就会被差评）。

3. 这些属性值有不同的量度。我们会在本章后面讨论特征缩放。

4. 最后，许多柱状图的尾巴很长：相较于左边，它们在中位数的右边延伸过远。对于某些机器学习算法，这会使检测规律变得更难些。我们会在后面尝试变换处理这些属性，使其变为正态分布。

希望你现在对要处理的数据有一定了解了。

> 警告：稍等！在你进一步查看数据之前，你需要创建一个测试集，将它放在一旁，千万不要再看它。

创建测试集

在这个阶段就分割数据，听起来很奇怪。毕竟，你只是简单快速地查看了数据而已，你需要再仔细调查下数据以决定使用什么算法。这么想是对的，但是人类的大脑是一个神奇的发现规律的系统，这意味着大脑非常容易发生过拟合：如果你查看了测试集，就会不经意地按照测试集中的规律来选择某个特定的机器学习模型。再当你使用测试集来评估误差率时，就会导致评估过于乐观，而实际部署的系统表现就会差。这称为数据透视偏差。

理论上，创建测试集很简单：只要随机挑选一些实例，一般是数据集的 20%，放到一边：

```
import numpy as np

def split_train_test(data, test_ratio):
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
```

然后可以像下面这样使用这个函数：

```
>>> train_set, test_set = split_train_test(housing, 0.2)
>>> print(len(train_set), "train +", len(test_set), "test")
16512 train + 4128 test
```

这个方法可行，但是并不完美：如果再次运行程序，就会产生一个不同的测试集！多次运行之后，你（或你的机器学习算法）就会得到整个数据集，这是需要避免的。

解决的办法之一是保存第一次运行得到的测试集，并在随后的过程加载。另一种方法是在调用`np.random.permutation()`之前，设置随机数生成器的种子（比如`np.random.seed(42)`），以产生总是相同的洗牌指数（shuffled indices）。

但是如果数据集更新，这两个方法都会失效。一个通常的解决办法是使用每个实例的ID来判定这个实例是否应该放入测试集（假设每个实例都有唯一并且不变的ID）。例如，你可以计算出每个实例ID的哈希值，只保留其最后一个字节，如果该值小于等于 51（约为 256 的

20%），就将其放入测试集。这样可以保证在多次运行中，测试集保持不变，即使更新了数据集。新的测试集会包含新实例中的 20%，但不会有之前位于训练集的实例。下面是一种可用的方法：

```
import hashlib

def test_set_check(identifier, test_ratio, hash):
    return hash(np.int64(identifier)).digest()[-1] < 256 * test_ratio

def split_train_test_by_id(data, test_ratio, id_column, hash=hashlib.md5):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_: test_set_check(id_, test_ratio, hash))
    return data.loc[~in_test_set], data.loc[in_test_set]
```

不过，房产数据集没有ID这一列。最简单的方法是使用行索引作为 ID：

```
housing_with_id = housing.reset_index()   # adds an `index` column
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "index")
```

如果使用行索引作为唯一识别码，你需要保证新数据都放到现有数据的尾部，且没有行被删除。如果做不到，则可以用最稳定的特征来创建唯一识别码。例如，一个区的维度和经度在几百万年之内是不变的，所以可以将两者结合成一个 ID：

```
housing_with_id["id"] = housing["longitude"] * 1000 + housing["latitude"]
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "id")
```

Scikit-Learn 提供了一些函数，可以用多种方式将数据集分割成多个子集。最简单的函数是 `train_test_split`，它的作用和之前的函数 `split_train_test` 很像，并带有其它一些功能。首先，它有一个 `random_state` 参数，可以设定前面讲过的随机生成器种子；第二，你可以将种子传递给多个行数相同的数据集，可以在相同的索引上分割数据集（这个功能非常有用，比如你的标签值是放在另一个 `DataFrame` 里的）：

```
from sklearn.model_selection import train_test_split

train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
```

目前为止，我们采用的都是纯随机的取样方法。当你的数据集很大时（尤其是和属性数相比），这通常可行；但如果数据集不大，就会有采样偏差的风险。当一个调查公司想要对 1000 个人进行调查，它们不是在电话亭里随机选 1000 个人出来。调查公司要保证这 1000 个人对人群整体有代表性。例如，美国人口的 51.3% 是女性，48.7% 是男性。所以在美国，严谨的调查需要保证样本也是这个比例：513 名女性，487 名男性。这称作分层采样（stratified sampling）：将人群分成均匀的子分组，称为分层，从每个分层去取合适数量的实例，以保证测试集对总人数有代表性。如果调查公司采用纯随机采样，会有 12% 的概率导致采样偏差：女性人数少于 49%，或多于 54%。不管发生那种情况，调查结果都会严重偏差。

假设专家告诉你，收入中位数是预测房价中位数非常重要的属性。你可能想要保证测试集可以代表整体数据集中的多种收入分类。因为收入中位数是一个连续的数值属性，你首先需要创建一个收入类别属性。再仔细地看一下收入中位数的柱状图（图 2-9）（译注：该图是对收入中位数处理过后的图）：

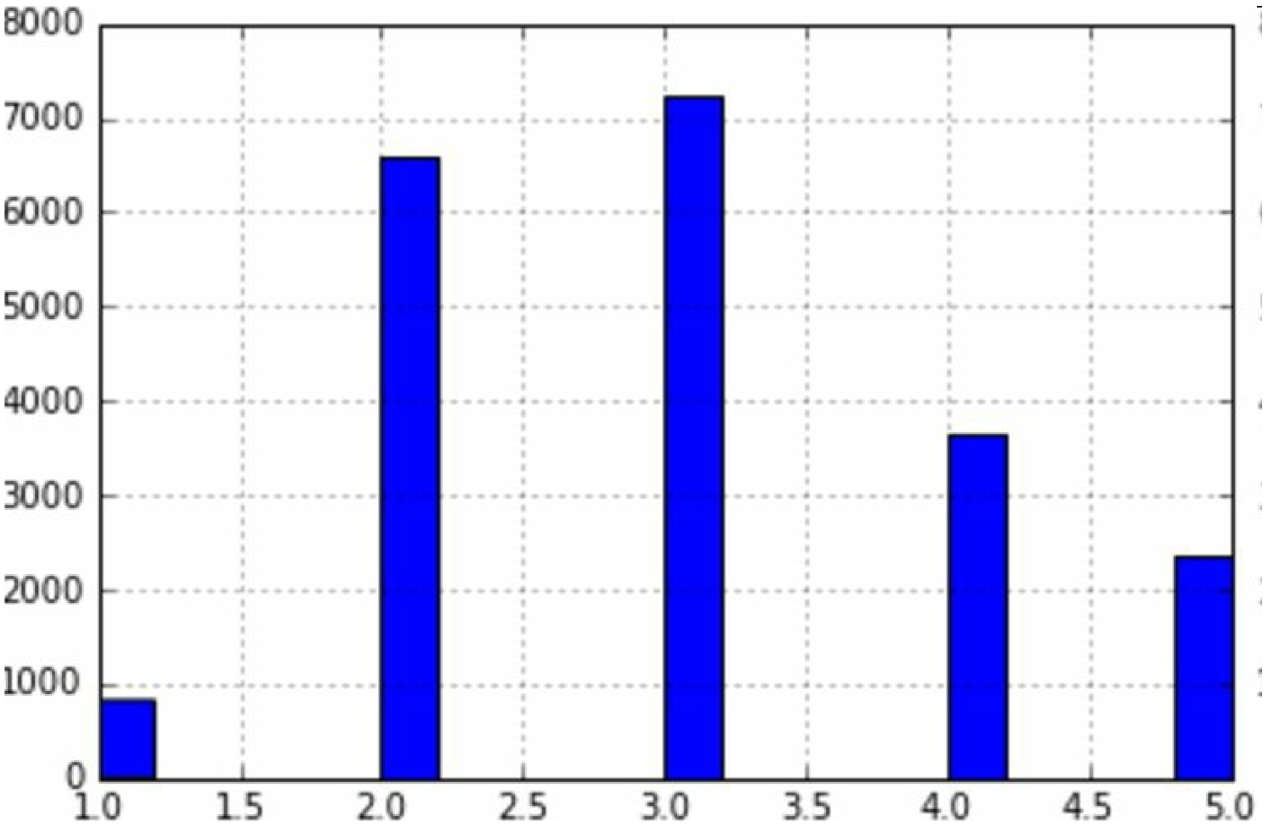


图 2-9 收入分类的柱状图

大多数的收入中位数的值聚集在 2-5（万美元），但是一些收入中位数会超过 6。数据集中的每个分层都要有足够的实例位于你的数据中，这点很重要。否则，对分层重要性的评估就会有偏差。这意味着，你不能有过多的分层，且每个分层都要足够大。后面的代码通过将收入中位数除以 1.5（以限制收入分类的数量），创建了一个收入类别属性，用 `ceil` 对值舍入（以产生离散的分类），然后将所有大于 5的分类归入到分类 5：

```
housing["income_cat"] = np.ceil(housing["median_income"] / 1.5)
housing["income_cat"].where(housing["income_cat"] < 5, 5.0, inplace=True)
```

现在，就可以根据收入分类，进行分层采样。你可以使用 Scikit-Learn 的 `StratifiedShuffleSplit` 类：

```
from sklearn.model_selection import StratifiedShuffleSplit

split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)

for train_index, test_index in split.split(housing, housing["income_cat"]):
    strat_train_set = housing.loc[train_index]
    strat_test_set = housing.loc[test_index]
```

检查下结果是否符合预期。你可以在完整的房产数据集中查看收入分类比例：

```
>>> housing["income_cat"].value_counts() / len(housing)
3.0    0.350581
2.0    0.318847
4.0    0.176308
5.0    0.114438
1.0    0.039826
Name: income_cat, dtype: float64
```

使用相似的代码，还可以测量测试集中收入分类的比例。图 2-10 对比了总数据集、分层采样的测试集、纯随机采样测试集的收入分类比例。可以看到，分层采样测试集的收入分类比例与总数据集几乎相同，而随机采样数据集偏差严重。

	Overall	Random	Stratified	Rand. %error	Strat. %error
1.0	0.039826	0.040213	0.039738	0.973236	-0.219137
2.0	0.318847	0.324370	0.318876	1.732260	0.009032
3.0	0.350581	0.358527	0.350618	2.266446	0.010408
4.0	0.176308	0.167393	0.176399	-5.056334	0.051717
5.0	0.114438	0.109496	0.114369	-4.318374	-0.060464

图 2-10 分层采样和纯随机采样的样本偏差比较

现在，你需要删除 `income_cat` 属性，使数据回到初始状态：

```
for set in (strat_train_set, strat_test_set):
    set.drop(["income_cat"], axis=1, inplace=True)
```

我们用了大量时间来生成测试集的原因是：测试集通常被忽略，但实际是机器学习非常重要的一部分。还有，生成测试集过程中的许多思路对于后面的交叉验证讨论是非常有帮助的。接下来进入下一阶段：数据探索。

数据探索和可视化、发现规律

目前为止，你只是快速查看了数据，对要处理的数据有了整体了解。现在的目标是更深的探索数据。

首先，保证你将测试集放在了一旁，只是研究训练集。另外，如果训练集非常大，你可能需要再采样一个探索集，保证操作方便快速。在我们的案例中，数据集很小，所以可以在全集上直接工作。创建一个副本，以免损伤训练集：

```
housing = strat_train_set.copy()
```

地理数据可视化

因为存在地理信息（纬度和经度），创建一个所有街区的散点图来数据可视化是一个不错的主意（图 2-11）：

```
housing.plot(kind="scatter", x="longitude", y="latitude")
```

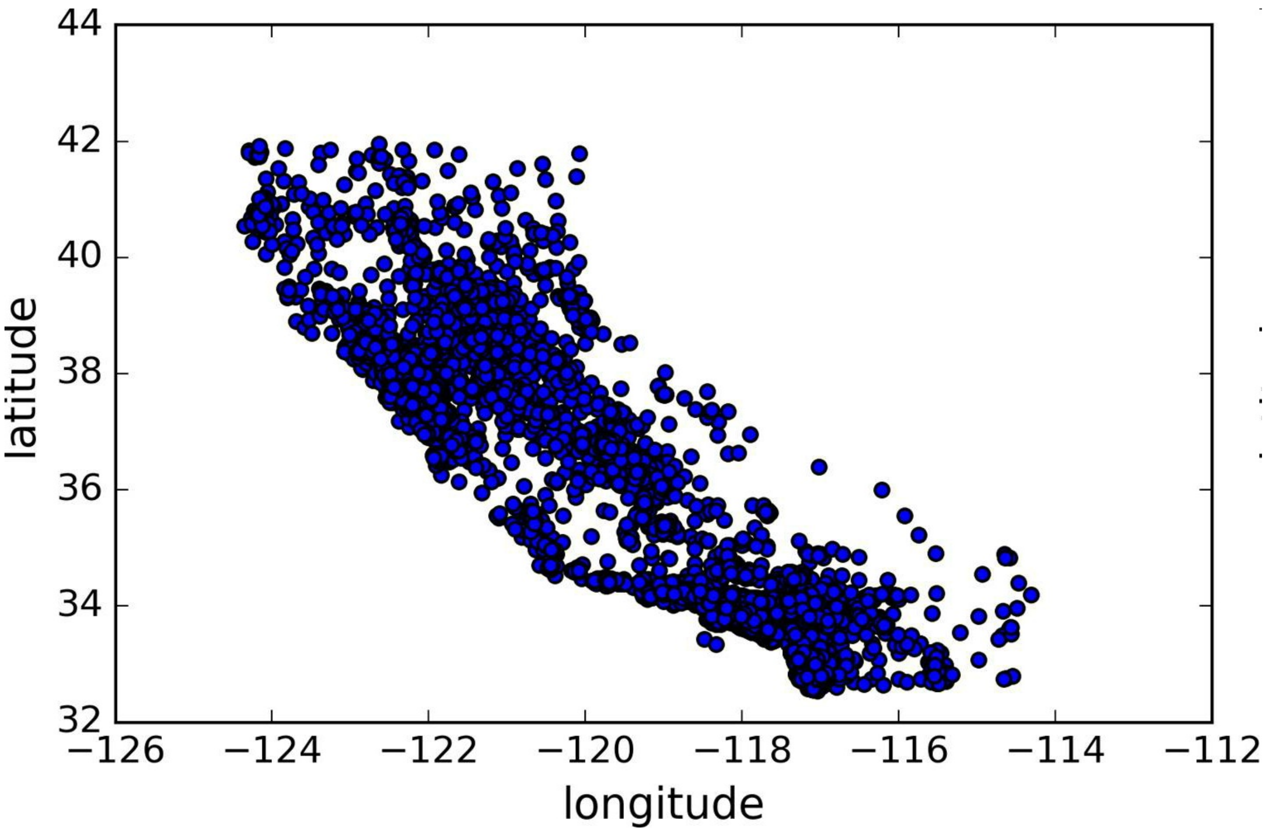


图 2-11 数据的地理信息散点图

这张图看起来很像加州，但是看不出什么特别的规律。将`alpha`设为 0.1，可以更容易看出数据点的密度（图 2-12）：

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)
```

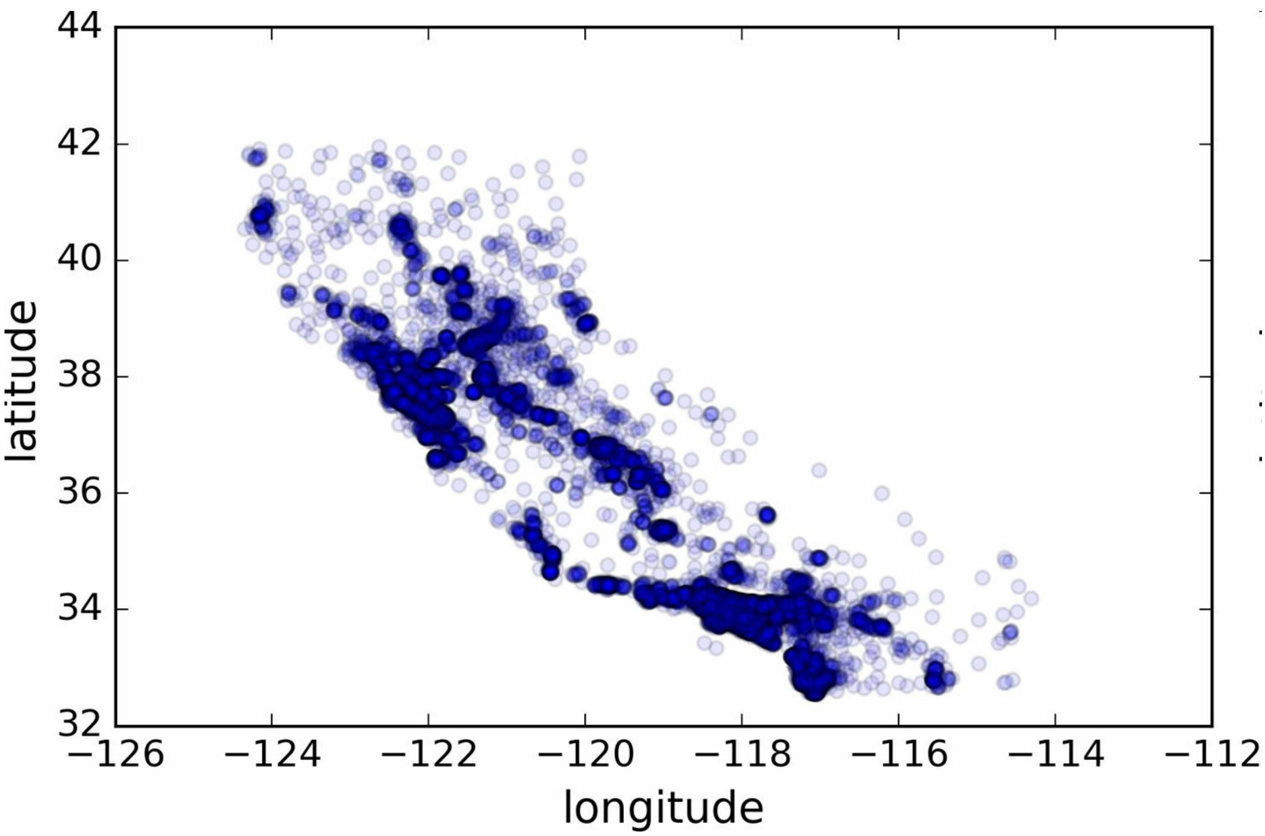


图 2-12 显示高密度区域的散点图

现在看起来好多了：可以非常清楚地看到高密度区域，湾区、洛杉矶和圣迭戈，以及中央谷，特别是从萨克拉门托和弗雷斯诺。

通常来讲，人类的大脑非常善于发现图片中的规律，但是需要调整可视化参数使规律显现出来。

现在来看房价（图 2-13）。每个圈的半径表示街区的人口（选项 `s` ），颜色代表价格（选项 `c` ）。我们用预先定义的名为 `jet` 的颜色图（选项 `cmap` ），它的范围是从蓝色（低价）到红色（高价）：

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,
    s=housing["population"]/100, label="population",
    c="median_house_value", cmap=plt.get_cmap("jet"), colorbar=True,
)
plt.legend()
```

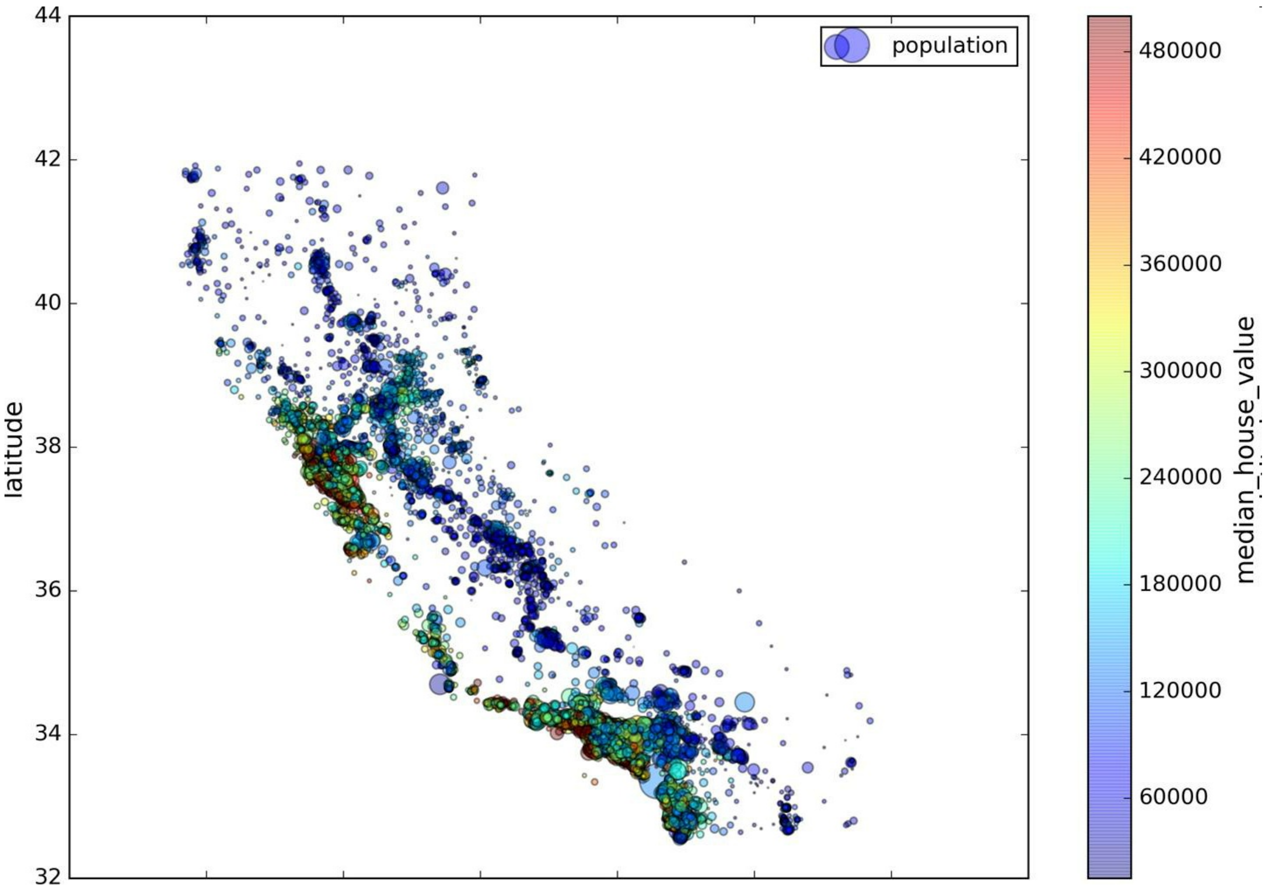


图 2-13 加州房价

这张图说明房价和位置（比如，靠海）和人口密度联系密切，这点你可能早就知道。可以使用聚类算法来检测主要的聚集，用一个新的特征值测量聚集中心的距离。尽管北加州海岸区域的房价不是非常高，但离大海距离属性也可能很有用，所以这不是用一个简单的规则就可以定义的问题。

查找关联

因为数据集并不是非常大，你可以很容易地使用 `corr()` 方法计算出每对属性间的标准相关系数（standard correlation coefficient，也称作皮尔逊相关系数）：

```
corr_matrix = housing.corr()
```

现在来看下每个属性和房价中位数的关联度：

```
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value    1.000000
median_income         0.687170
total_rooms           0.135231
housing_median_age    0.114220
households            0.064702
total_bedrooms        0.047865
population           -0.026699
longitude            -0.047279
latitude             -0.142826
Name: median_house_value, dtype: float64
```

相关系数的范围是 -1 到 1。当接近 1 时，意味强正相关；例如，当收入中位数增加时，房价中位数也会增加。当相关系数接近 -1 时，意味强负相关；你可以看到，纬度和房价中位数有轻微的负相关性（即，越往北，房价越可能降低）。最后，相关系数接近 0，意味没有线性相关性。图 2-14 展示了相关系数在横轴和纵轴之间的不同图形。

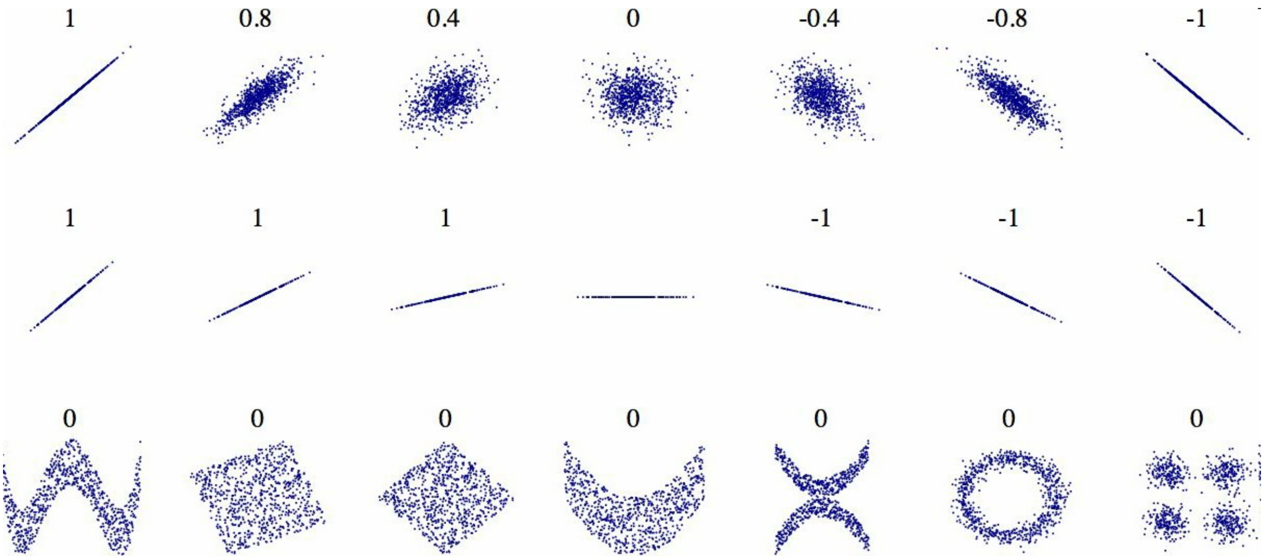


图 2-14 不同数据集的标准相关系数（来源：Wikipedia；公共领域图片）

> 警告：相关系数只测量线性关系（如果 `x` 上升，`y` 则上升或下降）。相关系数可能会完全忽略非线性关系（例如，如果 `x` 接近 0，则 `y` 值会变高）。在上面图片的最后一行中，他们的相关系数都接近于 0，尽管它们的轴并不独立：这些就是非线性关系的例子。另外，第二行的相关系数等于 1 或 -1；这和斜率没有任何关系。例如，你的身高（单位是英寸）与身高（单位是英尺或纳米）的相关系数就是 1。

另一种检测属性间相关系数的方法是使用 Pandas 的 `scatter_matrix` 函数，它能画出每个数值属性对每个其它数值属性的图。因为现在共有 11 个数值属性，你可以得到 `11 ** 2 = 121` 张图，在一页上画不下，所以只关注几个和房价中位数最有可能相关的属性（图 2-15）：

```
from pandas.tools.plotting import scatter_matrix

attributes = ["median_house_value", "median_income", "total_rooms",
              "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
```

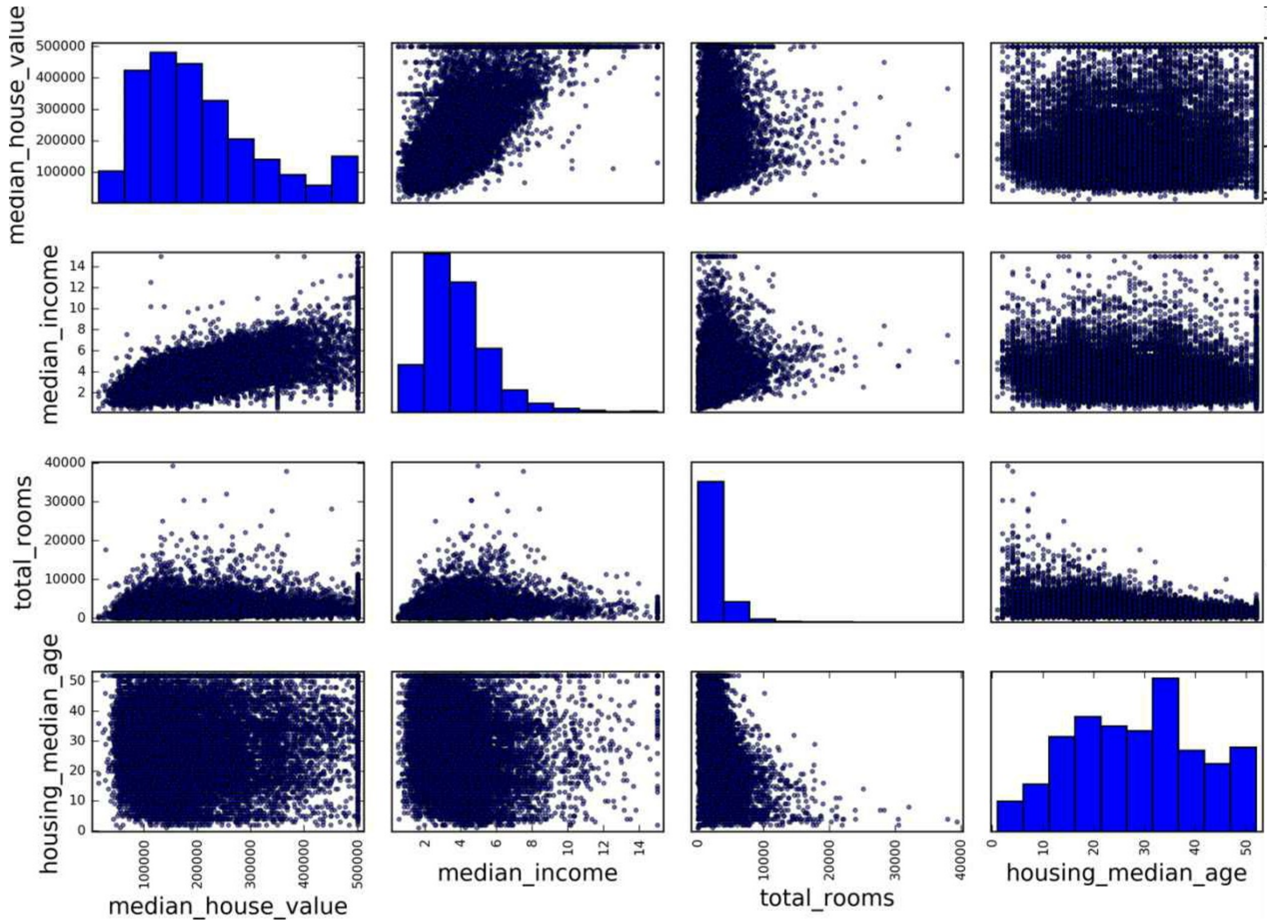


图 2-15 散点矩阵

如果 pandas 将每个变量对自己作图，主对角线（左上到右下）都会是直线图。所以 Pandas 展示的是每个属性的柱状图（也可以是其它的，请参考 Pandas 文档）。

最有希望用来预测房价中位数的属性是收入中位数，因此将这张图放大（图 2-16）：

```
housing.plot(kind="scatter", x="median_income",y="median_house_value",
             alpha=0.1)
```

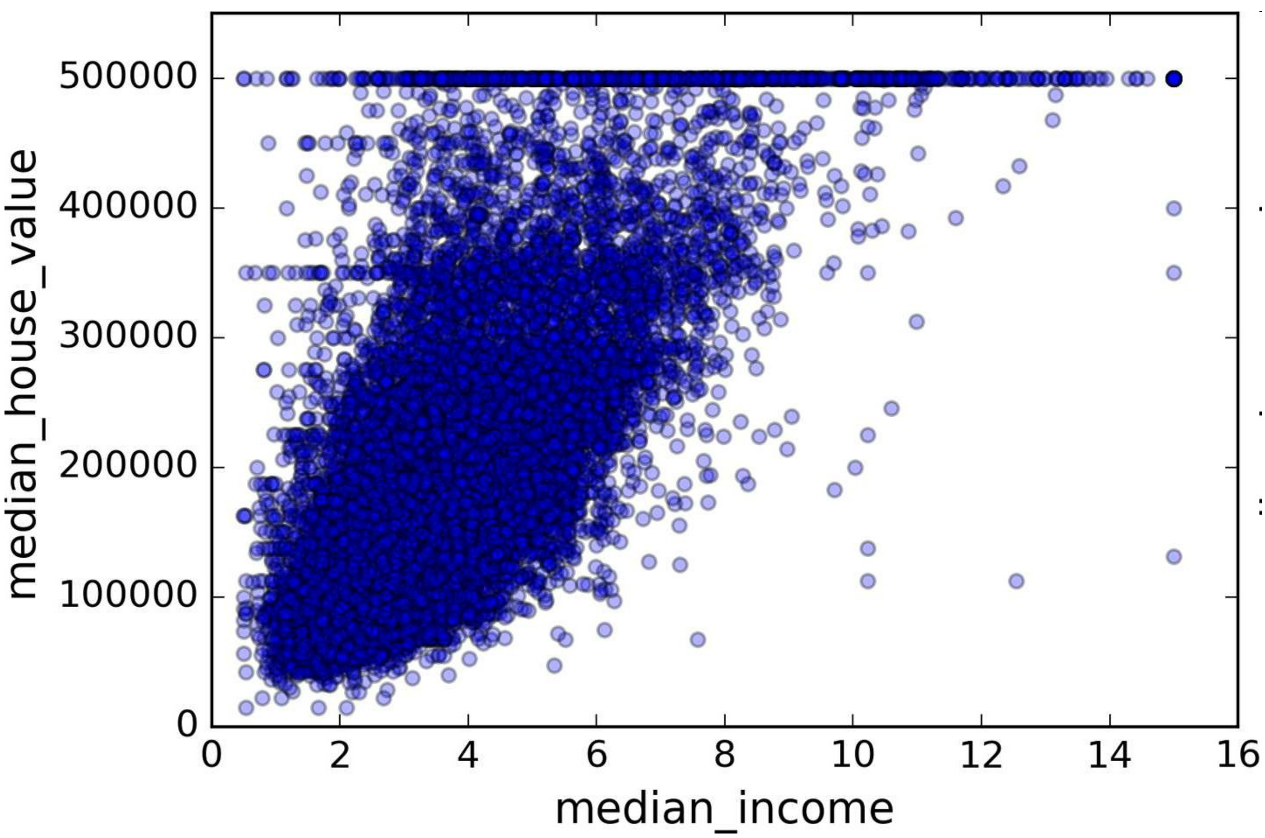


图 2-16 收入中位数 vs 房价中位数

这张图说明了几点。首先，相关性非常高；可以清晰地看到向上的趋势，并且数据点不是非常分散。第二，我们之前看到的最高价，清晰地呈现为一条位于 500000 美元的水平线。这张图也呈现了一些不是那么明显的直线：一条位于 450000 美元的直线，一条位于 350000 美元的直线，一条在 280000 美元的线，和一些更靠下的线。你可能希望去除对应的街区，以防止算法重复这些巧合。

属性组合试验

希望前面的一节能教给你一些探索数据、发现规律的方法。你发现了一些数据的巧合，需要在给算法提供数据之前，将其去除。你还发现了一些属性间有趣的关联，特别是目标属性。你还注意到一些属性具有长尾分布，因此你可能要将其进行转换（例如，计算其 `log` 对数）。当然，不同项目的处理方法各不相同，但大体思路是相似的。

给算法准备数据之前，你需要做的最后一件事是尝试多种属性组合。例如，如果你不知道某个街区有多少户，该街区的总房间数就没什么用。你真正需要的是每户有几个房间。相似的，总卧室数也不重要：你可能需要将其与房间数进行比较。每户的人口数也是一个有趣的属性组合。让我们来创建这些新的属性：

```
housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]
housing["population_per_household"]=housing["population"]/housing["households"]
```

现在，再来看相关矩阵：

```
>>> corr_matrix = housing.corr()
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value          1.000000
median_income               0.687170
rooms_per_household         0.199343
total_rooms                 0.135231
housing_median_age          0.114220
households                  0.064702
total_bedrooms              0.047865
population_per_household   -0.021984
population                 -0.026699
longitude                  -0.047279
latitude                   -0.142826
bedrooms_per_room          -0.260070
Name: median_house_value, dtype: float64
```

看起来不错！与总房间数或卧室数相比，新的 `bedrooms_per_room` 属性与房价中位数的关联更强。显然，卧室数/总房间数的比例越低，房价就越高。每户的房间数也比街区的总房间数的更有信息，很明显，房屋越大，房价就越高。

这一步的数据探索不必非常完备，此处的目的是有一个正确的开始，快速发现规律，以得到一个合理的原型。但是这是一个交互过程：一旦你得到了一个原型，并运行起来，你就可以分析它的输出，进而发现更多的规律，然后再回到数据探索这步。

为机器学习算法准备数据

现在来为机器学习算法准备数据。不要手工来做，你需要写一些函数，理由如下：

- 函数可以让你在任何数据集上（比如，你下一次获取的是一个新的数据集）方便地进行重复数据转换。
- 你能慢慢建立一个转换函数库，可以在未来的项目中复用。
- 在将数据传给算法之前，你可以在实时系统中使用这些函数。
- 这可以让你方便地尝试多种数据转换，查看哪些转换方法结合起来效果最好。

但是，还是先回到干净的训练集（通过再次复制 `strat_train_set` ），将预测量和标签分开，因为我们不想对预测量和目标值应用相同的转换（注意 `drop()` 创建了一份数据的备份，而不影响 `strat_train_set` ）：

```
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
```

数据清洗

大多机器学习算法不能处理缺失的特征，因此先创建一些函数来处理特征缺失的问题。前面，你应该注意到了属性 `total_bedrooms` 有一些缺失值。有三个解决选项：

- 去掉对应的街区；
- 去掉整个属性；
- 进行赋值（0、平均值、中位数等等）。

用 `DataFrame` 的 `dropna()`，`drop()`，和 `fillna()` 方法，可以方便地实现：

```
housing.dropna(subset=["total_bedrooms"])    # 选项1
housing.drop("total_bedrooms", axis=1)       # 选项2
median = housing["total_bedrooms"].median()
housing["total_bedrooms"].fillna(median)     # 选项3
```

如果选择选项 3，你需要计算训练集的中位数，用中位数填充训练集的缺失值，不要忘记保存该中位数。后面用测试集评估系统时，需要替换测试集中的缺失值，也可以用来实时替换新数据中的缺失值。

Scikit-Learn 提供了一个方便的类来处理缺失值：`Imputer`。下面是其使用方法：首先，需要创建一个 `Imputer` 实例，指定用某属性的中位数来替换该属性所有的缺失值：

```
from sklearn.preprocessing import Imputer

imputer = Imputer(strategy="median")
```

因为只有数值属性才能算出中位数，我们需要创建一份不包括文本属性 `ocean_proximity` 的数据副本：

```
housing_num = housing.drop("ocean_proximity", axis=1)
```

现在，就可以用 `fit()` 方法将 `imputer` 实例拟合到训练数据：

```
imputer.fit(housing_num)
```

`imputer` 计算出了每个属性的中位数，并将结果保存在了实例变量 `statistics_` 中。虽然此时只有属性 `total_bedrooms` 存在缺失值，但我们不能确定在以后的新的数据中会不会有其他属性也存在缺失值，所以安全的做法是将 `imputer` 应用到每个数值：

```
>>> imputer.statistics_
array([ -118.51 , 34.26 , 29. , 2119. , 433. , 1164. , 408. , 3.5414])
>>> housing_num.median().values
array([ -118.51 , 34.26 , 29. , 2119. , 433. , 1164. , 408. , 3.5414])
```

现在，你就可以使用这个“训练过的” `imputer` 来对训练集进行转换，将缺失值替换为中位数：

```
X = imputer.transform(housing_num)
```

结果是一个包含转换后特征的普通的 Numpy 数组。如果你想将其放回到 Pandas `DataFrame` 中，也很简单：

```
housing_tr = pd.DataFrame(X, columns=housing_num.columns)
```

> Scikit-Learn 设计
>
> Scikit-Learn 设计的 API 设计的非常好。它的主要设计原则是：
>
> - 一致性：所有对象的接口一致且简单：
>   - 估计器（estimator）。任何可以基于数据集对一些参数进行估计的对象都被称为估计器（比如，`imputer`就是个估计器）。估计本身是通过`fit()`方法，只需要一个数据集作为参数（对于监督学习算法，需要两个数据集；第二个数据集包含标签）。任何其它用来指导估计过程的参数都被当做超参数（比如`imputer`的`strategy`），并且超参数要被设置成实例变量（通常通过构造器参数设置）。
>   - 转换器（transformer）。一些估计器（比如`imputer`）也可以转换数据集，这些估计器被称为转换器。API也是相当简单：转换是通过`transform()`方法，被转换的数据集作为参数。返回的是经过转换的数据集。转换过程依赖学习到的参数，比如`imputer`的例子。所有的转换都有一个便捷的方法`fit_transform()`，等同于调用`fit()`再`transform()`（但有时`fit_transform()`经过优化，运行的更快）。
>   - 预测器（predictor）。最后，一些估计器可以根据给出的数据集做预测，这些估计器称为预测器。例如，上一章的`LinearRegression`模型就是一个预测器：它根据一个国家的人均 GDP 预测生活满意度。预测器有一个`predict()`方法，可以用新实例的数据集做出相应的预测。预测器还有一个`score()`方法，可以根据测试集（和相应的标签，如果是监督学习算法的话）对预测进行衡器。
> - 可检验。所有估计器的超参数都可以通过实例的public变量直接访问（比如，`imputer.strategy`），并且所有估计器学习到的参数也可以通过在实例变量名后加下划线来访问（比如，`imputer.statistics_`）。
>
> - 类不可扩散。数据集被表示成 NumPy 数组或 SciPy 稀疏矩阵，而不是自制的类。超参数只是普通的 Python 字符串或数字。
>
> - 可组合。尽可能使用现存的模块。例如，用任意的转换器序列加上一个估计器，就可以做成一个流水线，后面会看到例子。
>
> - 合理的默认值。Scikit-Learn 给大多数参数提供了合理的默认值，很容易就能创建一个系统。

处理文本和类别属性

前面，我们丢弃了类别属性`ocean_proximity`，因为它是一个文本属性，不能计算出中位数。大多数机器学习算法跟喜欢和数字打交道，所以让我们把这些文本标签转换为数字。

Scikit-Learn 为这个任务提供了一个转换器`LabelEncoder`：

```
>>> from sklearn.preprocessing import LabelEncoder
>>> encoder = LabelEncoder()
>>> housing_cat = housing["ocean_proximity"]
>>> housing_cat_encoded = encoder.fit_transform(housing_cat)
>>> housing_cat_encoded
array([1, 1, 4, ..., 1, 0, 3])
```

> 译注:
>
> 在原书中使用`LabelEncoder`转换器来转换文本特征列的方式是错误的，该转换器只能用来转换标签（正如其名）。在这里使用`LabelEncoder`没有出错的原因是该数据只有一列文本特征值，在有多个文本特征列的时候就会出错。应使用`factorize()`方法来进行操作：
>
> ```
> housing_cat_encoded, housing_categories = housing_cat.factorize()
> housing_cat_encoded[:10]
> ```

好了一些，现在就可以在任何 ML 算法里用这个数值数据了。你可以查看映射表，编码器是通过属性`classes_`来学习的（`<1H OCEAN`被映射为 0，`INLAND`被映射为 1，等等）：

```
>>> print(encoder.classes_)
['<1H OCEAN' 'INLAND' 'ISLAND' 'NEAR BAY' 'NEAR OCEAN']
```

这种做法的问题是，ML 算法会认为两个临近的值比两个疏远的值要更相似。显然这样不对（比如，分类 0 和 4 比 0 和 1 更相似）。要解决这个问题，一个常见的方法是给每个分类创建一个二元属性：当分类是`<1H OCEAN`，该属性为 1（否则为 0），当分类是`INLAND`，另一个属性等于 1（否则为 0），以此类推。这称作独热编码（One-Hot Encoding），因为只有一个属性会等于 1（热），其余会是 0（冷）。

Scikit-Learn 提供了一个编码器`OneHotEncoder`，用于将整数分类值转变为独热向量。注意`fit_transform()`用于 2D 数组，而`housing_cat_encoded`是一个 1D 数组，所以需要将其变形：

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> encoder = OneHotEncoder()
>>> housing_cat_1hot = encoder.fit_transform(housing_cat_encoded.reshape(-1,1))
>>> housing_cat_1hot
<16513x5 sparse matrix of type '<class 'numpy.float64'>'
    with 16513 stored elements in Compressed Sparse Row format>
```

注意输出结果是一个 SciPy 稀疏矩阵，而不是 NumPy 数组。当类别属性有数千个分类时，这样非常有用。经过独热编码，我们得到了一个有数千列的矩阵，这个矩阵每行只有一个 1，其余都是 0。使用大量内存来存储这些 0 非常浪费，所以稀疏矩阵只存储非零元素的位置。你可以像一个 2D 数据那样进行使用，但是如果你真的想将其转变成一个（密集的）NumPy 数组，只需调用 `toarray()` 方法：

```
>>> housing_cat_1hot.toarray()
array([[ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  1.],
       ...,
       [ 0.,  1.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.]])
```

使用类 `LabelBinarizer` ，我们可以用一步执行这两个转换（从文本分类到整数分类，再从整数分类到独热向量）：

```
>>> from sklearn.preprocessing import LabelBinarizer
>>> encoder = LabelBinarizer()
>>> housing_cat_1hot = encoder.fit_transform(housing_cat)
>>> housing_cat_1hot
array([[0, 1, 0, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 0, 0, 0, 1],
       ...,
       [0, 1, 0, 0, 0],
       [1, 0, 0, 0, 0],
       [0, 0, 0, 1, 0]])
```

注意默认返回的结果是一个密集 NumPy 数组。向构造器 `LabelBinarizer` 传递 `sparse_output=True` ，就可以得到一个稀疏矩阵。

> 译注:
>
> 在原书中使用 `LabelBinarizer` 的方式也是错误的，该类也应用于标签列的转换。正确做法是使用sklearn即将提供的 `CategoricalEncoder` 类。如果在你阅读此文时sklearn中尚未提供此类，用如下方式代替：（来自Pull Request #9151）
>
> ```
> # Definition of the CategoricalEncoder class, copied from PR #9151.
> # Just run this cell, or copy it to your code, do not try to understand it (yet).
>
> from sklearn.base import BaseEstimator, TransformerMixin
> from sklearn.utils import check_array
> from sklearn.preprocessing import LabelEncoder
> from scipy import sparse
>
> class CategoricalEncoder(BaseEstimator, TransformerMixin):
>     """Encode categorical features as a numeric array.
>     The input to this transformer should be a matrix of integers or strings,
>     denoting the values taken on by categorical (discrete) features.
>     The features can be encoded using a one-hot aka one-of-K scheme
>     (``encoding='onehot'``, the default) or converted to ordinal integers
>     (``encoding='ordinal'``).
>     This encoding is needed for feeding categorical data to many scikit-learn
>     estimators, notably linear models and SVMs with the standard kernels.
>     Read more in the :ref:`User Guide <preprocessing_categorical_features>`.
> ```

```
    Parameters
    ----------
    encoding : str, 'onehot', 'onehot-dense' or 'ordinal'
        The type of encoding to use (default is 'onehot'):
        - 'onehot': encode the features using a one-hot aka one-of-K scheme
          (or also called 'dummy' encoding). This creates a binary column for
          each category and returns a sparse matrix.
        - 'onehot-dense': the same as 'onehot' but returns a dense array
          instead of a sparse matrix.
        - 'ordinal': encode the features as ordinal integers. This results in
          a single column of integers (0 to n_categories - 1) per feature.
    categories : 'auto' or a list of lists/arrays of values.
        Categories (unique values) per feature:
        - 'auto' : Determine categories automatically from the training data.
        - list : ``categories[i]`` holds the categories expected in the ith
          column. The passed categories are sorted before encoding the data
          (used categories can be found in the ``categories_`` attribute).
    dtype : number type, default np.float64
        Desired dtype of output.
    handle_unknown : 'error' (default) or 'ignore'
        Whether to raise an error or ignore if a unknown categorical feature is
        present during transform (default is to raise). When this is parameter
        is set to 'ignore' and an unknown category is encountered during
        transform, the resulting one-hot encoded columns for this feature
        will be all zeros.
        Ignoring unknown categories is not supported for
        ``encoding='ordinal'``.
    Attributes
    ----------
    categories_ : list of arrays
        The categories of each feature determined during fitting. When
        categories were specified manually, this holds the sorted categories
        (in order corresponding with output of `transform`).
    Examples
    --------
    Given a dataset with three features and two samples, we let the encoder
    find the maximum value per feature and transform the data to a binary
    one-hot encoding.
    >>> from sklearn.preprocessing import CategoricalEncoder
    >>> enc = CategoricalEncoder(handle_unknown='ignore')
    >>> enc.fit([[0, 0, 3], [1, 1, 0], [0, 2, 1], [1, 0, 2]])
    ... # doctest: +ELLIPSIS
    CategoricalEncoder(categories='auto', dtype=<... 'numpy.float64'>,
              encoding='onehot', handle_unknown='ignore')
    >>> enc.transform([[0, 1, 1], [1, 0, 4]]).toarray()
    array([[ 1.,  0.,  0.,  1.,  0.,  0.,  1.,  0.,  0.],
           [ 0.,  1.,  1.,  0.,  0.,  0.,  0.,  0.,  0.]])
    See also
    --------
    sklearn.preprocessing.OneHotEncoder : performs a one-hot encoding of
      integer ordinal features. The ``OneHotEncoder assumes`` that input
      features take on values in the range ``[0, max(feature)]`` instead of
      using the unique values.
    sklearn.feature_extraction.DictVectorizer : performs a one-hot encoding of
      dictionary items (also handles string-valued features).
    sklearn.feature_extraction.FeatureHasher : performs an approximate one-hot
      encoding of dictionary items or strings.
    """

    def __init__(self, encoding='onehot', categories='auto', dtype=np.float64,
                 handle_unknown='error'):
        self.encoding = encoding
        self.categories = categories
        self.dtype = dtype
        self.handle_unknown = handle_unknown

    def fit(self, X, y=None):
        """Fit the CategoricalEncoder to X.
        Parameters
        ----------
        X : array-like, shape [n_samples, n_feature]
            The data to determine the categories of each feature.
```

```
        Returns
        -------
        self
        """

        if self.encoding not in ['onehot', 'onehot-dense', 'ordinal']:
            template = ("encoding should be either 'onehot', 'onehot-dense' "
                        "or 'ordinal', got %s")
            raise ValueError(template % self.handle_unknown)

        if self.handle_unknown not in ['error', 'ignore']:
            template = ("handle_unknown should be either 'error' or "
                        "'ignore', got %s")
            raise ValueError(template % self.handle_unknown)

        if self.encoding == 'ordinal' and self.handle_unknown == 'ignore':
            raise ValueError("handle_unknown='ignore' is not supported for"
                             " encoding='ordinal'")

        X = check_array(X, dtype=np.object, accept_sparse='csc', copy=True)
        n_samples, n_features = X.shape

        self._label_encoders_ = [LabelEncoder() for _ in range(n_features)]

        for i in range(n_features):
            le = self._label_encoders_[i]
            Xi = X[:, i]
            if self.categories == 'auto':
                le.fit(Xi)
            else:
                valid_mask = np.in1d(Xi, self.categories[i])
                if not np.all(valid_mask):
                    if self.handle_unknown == 'error':
                        diff = np.unique(Xi[~valid_mask])
                        msg = ("Found unknown categories {0} in column {1}"
                               " during fit".format(diff, i))
                        raise ValueError(msg)
                le.classes_ = np.array(np.sort(self.categories[i]))

        self.categories_ = [le.classes_ for le in self._label_encoders_]

        return self

    def transform(self, X):
        """Transform X using one-hot encoding.
        Parameters
        ----------
        X : array-like, shape [n_samples, n_features]
            The data to encode.
        Returns
        -------
        X_out : sparse matrix or a 2-d array
            Transformed input.
        """
        X = check_array(X, accept_sparse='csc', dtype=np.object, copy=True)
        n_samples, n_features = X.shape
        X_int = np.zeros_like(X, dtype=np.int)
        X_mask = np.ones_like(X, dtype=np.bool)

        for i in range(n_features):
            valid_mask = np.in1d(X[:, i], self.categories_[i])

            if not np.all(valid_mask):
                if self.handle_unknown == 'error':
                    diff = np.unique(X[~valid_mask, i])
                    msg = ("Found unknown categories {0} in column {1}"
                           " during transform".format(diff, i))
                    raise ValueError(msg)
                else:
                    # Set the problematic rows to an acceptable value and
                    # continue `The rows are marked `X_mask` and will be
                    # removed later.
```

```
                    X_mask[:, i] = valid_mask
                    X[:, i][~valid_mask] = self.categories_[i][0]
            X_int[:, i] = self._label_encoders_[i].transform(X[:, i])

        if self.encoding == 'ordinal':
            return X_int.astype(self.dtype, copy=False)

        mask = X_mask.ravel()
        n_values = [cats.shape[0] for cats in self.categories_]
        n_values = np.array([0] + n_values)
        indices = np.cumsum(n_values)

        column_indices = (X_int + indices[:-1]).ravel()[mask]
        row_indices = np.repeat(np.arange(n_samples, dtype=np.int32),
                                n_features)[mask]
        data = np.ones(n_samples * n_features)[mask]

        out = sparse.csc_matrix((data, (row_indices, column_indices)),
                                shape=(n_samples, indices[-1]),
                                dtype=self.dtype).tocsr()
        if self.encoding == 'onehot-dense':
            return out.toarray()
        else:
            return out
```

转换方法：

```
#from sklearn.preprocessing import CategoricalEncoder # in future versions of Scikit-Learn

cat_encoder = CategoricalEncoder()
housing_cat_reshaped = housing_cat.values.reshape(-1, 1)
housing_cat_1hot = cat_encoder.fit_transform(housing_cat_reshaped)
housing_cat_1hot
```

自定义转换器

尽管 Scikit-Learn 提供了许多有用的转换器，你还是需要自己动手写转换器执行任务，比如自定义的清理操作，或属性组合。你需要让自制的转换器与 Scikit-Learn 组件（比如流水线）无缝衔接工作，因为 Scikit-Learn 是依赖鸭子类型的（而不是继承），你所需要做的是创建一个类并执行三个方法： `fit()` （返回 `self` ）， `transform()` ，和 `fit_transform()` 。通过添加 `TransformerMixin` 作为基类，可以很容易地得到最后一个。另外，如果你添加 `BaseEstimator` 作为基类（且构造器中避免使用 `*args` 和 `**kargs` ），你就能得到两个额外的方法（ `get_params()` 和 `set_params()` ），二者可以方便地进行超参数自动微调。例如，一个小转换器类添加了上面讨论的属性：

```
from sklearn.base import BaseEstimator, TransformerMixin
rooms_ix, bedrooms_ix, population_ix, household_ix = 3, 4, 5, 6

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True): # no *args or **kargs
        self.add_bedrooms_per_room = add_bedrooms_per_room
    def fit(self, X, y=None):
        return self  # nothing else to do
    def transform(self, X, y=None):
        rooms_per_household = X[:, rooms_ix] / X[:, household_ix]
        population_per_household = X[:, population_ix] / X[:, household_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
            return np.c_[X, rooms_per_household, population_per_household,
                         bedrooms_per_room]
        else:
            return np.c_[X, rooms_per_household, population_per_household]

attr_adder = CombinedAttributesAdder(add_bedrooms_per_room=False)
housing_extra_attribs = attr_adder.transform(housing.values)
```

在这个例子中，转换器有一个超参数 `add_bedrooms_per_room` ，默认设为 `True` （提供一个合理的默认值很有帮助）。这个超参数可以让你方便地发现添加了这个属性是否对机器学习算法有帮助。更一般地，你可以为每个不能完全确保的数据准备步骤添加一个超参数。数据准备步骤越自动化，可以自动化的操作组合就越多，越容易发现更好用的组合（并能节省大量时间）。

特征缩放

数据要做的最重要的转换之一是特征缩放。除了个别情况，当输入的数值属性量度不同时，机器学习算法的性能都不会好。这个规律也适用于房产数据：总房间数分布范围是 6 到 39320，而收入中位数只分布在 0 到 15。注意通常情况下我们不需要对目标值进行缩放。

有两种常见的方法可以让所有的属性有相同的量度：线性函数归一化（Min-Max scaling）和标准化（standardization）。

线性函数归一化（许多人称其为归一化（normalization））很简单：值被转变、重新缩放，直到范围变成 0 到 1。我们通过减去最小值，然后再除以最大值与最小值的差值，来进行归一化。Scikit-Learn 提供了一个转换器 `MinMaxScaler` 来实现这个功能。它有一个超参数 `feature_range` ，可以让你改变范围，如果不希望范围是 0 到 1。

标准化就很不同：首先减去平均值（所以标准化值的平均值总是 0），然后除以方差，使得到的分布具有单位方差。与归一化不同，标准化不会限定值到某个特定的范围，这对某些算法可能构成问题（比如，神经网络常需要输入值得范围是 0 到 1）。但是，标准化受到异常值的影响很小。例如，假设一个街区的收入中位数由于某种错误变成了100，归一化会将其它范围是 0 到 15 的值变为 0-0.15，但是标准化不会受什么影响。Scikit-Learn 提供了一个转换器 `StandardScaler` 来进行标准化。

> 警告：与所有的转换一样，缩放器只能向训练集拟合，而不是向完整的数据集（包括测试集）。只有这样，你才能用缩放器转换训练集和测试集（和新数据）。

转换流水线

你已经看到，存在许多数据转换步骤，需要按一定的顺序执行。幸运的是，Scikit-Learn 提供了类 `Pipeline`，来进行这一系列的转换。下面是一个数值属性的小流水线：

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
        ('imputer', Imputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
        ])

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

`Pipeline` 构造器需要一个定义步骤顺序的名字/估计器对的列表。除了最后一个估计器，其余都要是转换器（即，它们都要有 `fit_transform()` 方法）。名字可以随意起。

当你调用流水线的 `fit()` 方法，就会对所有转换器顺序调用 `fit_transform()` 方法，将每次调用的输出作为参数传递给下一个调用，一直到最后一个估计器，它只执行 `fit()` 方法。

流水线暴露相同的方法作为最终的估计器。在这个例子中，最后的估计器是一个 `StandardScaler`，它是一个转换器，因此这个流水线有一个 `transform()` 方法，可以顺序对数据做所有转换（它还有一个 `fit_transform` 方法可以使用，就不必先调用 `fit()` 再进行 `transform()`）。

你现在就有了一个对数值的流水线，你还需要对分类值应用 `LabelBinarizer`：如何将这些转换写成一个流水线呢？Scikit-Learn 提供了一个类 `FeatureUnion` 实现这个功能。你给它一列转换器（可以是所有的转换器），当调用它的 `transform()` 方法，每个转换器的 `transform()` 会被并行执行，等待输出，然后将输出合并起来，并返回结果（当然，调用它的 `fit()` 方法就会调用每个转换器的 `fit()`）。一个完整的处理数值和类别属性的流水线如下所示：

```
from sklearn.pipeline import FeatureUnion

num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]

num_pipeline = Pipeline([
        ('selector', DataFrameSelector(num_attribs)),
        ('imputer', Imputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])

cat_pipeline = Pipeline([
        ('selector', DataFrameSelector(cat_attribs)),
        ('label_binarizer', LabelBinarizer()),
    ])

full_pipeline = FeatureUnion(transformer_list=[
        ("num_pipeline", num_pipeline),
        ("cat_pipeline", cat_pipeline),
    ])
```

> 译注：
>
> 如果你在上面代码中的 `cat_pipeline` 流水线使用 `LabelBinarizer` 转换器会导致执行错误，解决方案是用上文提到的 `CategoricalEncoder` 转换器来代替：
>
> ```
> cat_pipeline = Pipeline([
>         ('selector', DataFrameSelector(cat_attribs)),
>         ('cat_encoder', CategoricalEncoder(encoding="onehot-dense")),
>     ])
> ```

你可以很简单地运行整个流水线：

```
>>> housing_prepared = full_pipeline.fit_transform(housing)
>>> housing_prepared
array([[ 0.73225807, -0.67331551,  0.58426443, ...,  0.        ,
         0.        ,  0.        ],
       [-0.99102923,  1.63234656, -0.92655887, ...,  0.        ,
         0.        ,  0.        ],
       [...]
>>> housing_prepared.shape
(16513, 17)
```

每个子流水线都以一个选择转换器开始：通过选择对应的属性（数值或分类）、丢弃其它的，来转换数据，并将输出 `DataFrame` 转变成一个 NumPy 数组。Scikit-Learn 没有工具来处理 Pandas `DataFrame` ，因此我们需要写一个简单的自定义转换器来做这项工作：

```
from sklearn.base import BaseEstimator, TransformerMixin

class DataFrameSelector(BaseEstimator, TransformerMixin):
    def __init__(self, attribute_names):
        self.attribute_names = attribute_names
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        return X[self.attribute_names].values
```

选择并训练模型

可到这一步了！你在前面限定了问题、获得了数据、探索了数据、采样了一个测试集、写了自动化的转换流水线来清理和为算法准备数据。现在，你已经准备好选择并训练一个机器学习模型了。

在训练集上训练和评估

好消息是基于前面的工作，接下来要做的比你想的要简单许多。像前一章那样，我们先来训练一个线性回归模型：

```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(housing_prepared, housing_labels)
```

完毕！你现在就有了一个可用的线性回归模型。用一些训练集中的实例做下验证：

```
>>> some_data = housing.iloc[:5]
>>> some_labels = housing_labels.iloc[:5]
>>> some_data_prepared = full_pipeline.transform(some_data)
>>> print("Predictions:\t", lin_reg.predict(some_data_prepared))
Predictions:     [ 303104.   44800.   308928.   294208.   368704.]
>>> print("Labels:\t\t", list(some_labels))
Labels:         [359400.0, 69700.0, 302100.0, 301300.0, 351900.0]
```

行的通，尽管预测并不怎么准确（比如，第二个预测偏离了 50%！）。让我们使用 Scikit-Learn 的 `mean_squared_error` 函数，用全部训练集来计算下这个回归模型的 RMSE：

```
>>> from sklearn.metrics import mean_squared_error
>>> housing_predictions = lin_reg.predict(housing_prepared)
>>> lin_mse = mean_squared_error(housing_labels, housing_predictions)
>>> lin_rmse = np.sqrt(lin_mse)
>>> lin_rmse
68628.413493824875
```

OK，有总比没有强，但显然结果并不好：大多数街区的 `median_housing_values` 位于 120000 到 265000 美元之间，因此预测误差 68628 美元不能让人满意。这是一个模型欠拟合训练数据的例子。当这种情况发生时，意味着特征没有提供足够多的信息来做出一个好的预测，或者模型并不强大。就像前一章看到的，修复欠拟合的主要方法是选择一个更强大的模型，给训练算法提供更好的特征，或去掉模型上的限制。这个模型还没有正则化，所以排除了最后一个选项。你可以尝试添加更多特征（比如，人口的对数值），但是首先让我们尝试一个更为复杂的模型，看看效果。

来训练一个 `DecisionTreeRegressor`。这是一个强大的模型，可以发现数据中复杂的非线性关系（决策树会在第 6 章详细讲解）。代码看起来很熟悉：

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor()
tree_reg.fit(housing_prepared, housing_labels)
```

现在模型就训练好了，用训练集评估下：

```
>>> housing_predictions = tree_reg.predict(housing_prepared)
>>> tree_mse = mean_squared_error(housing_labels, housing_predictions)
>>> tree_rmse = np.sqrt(tree_mse)
>>> tree_rmse
0.0
```

等一下，发生了什么？没有误差？这个模型可能是绝对完美的吗？当然，更大可能性是这个模型严重过拟合数据。如何确定呢？如前所述，直到你准备运行一个具备足够信心的模型，都不要碰测试集，因此你需要使用训练集的部分数据来做训练，用一部分来做模型验证。

使用交叉验证做更佳的评估

评估决策树模型的一种方法是用函数 `train_test_split` 来分割训练集，得到一个更小的训练集和一个验证集，然后用更小的训练集来训练模型，用验证集来评估。这需要一定工作量，并不难而且也可行。

另一种更好的方法是使用 Scikit-Learn 的交叉验证功能。下面的代码采用了 K 折交叉验证（K-fold cross-validation）：它随机地将训练集分成十个不同的子集，成为“折”，然后训练评估决策树模型 10 次，每次选一个不用的折来做评估，用其它 9 个来做训练。结果是一个包含 10 个评分的数组：

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
                         scoring="neg_mean_squared_error", cv=10)
rmse_scores = np.sqrt(-scores)
```

> 警告：Scikit-Learn 交叉验证功能期望的是效用函数（越大越好）而不是损失函数（越低越好），因此得分函数实际上与 MSE 相反（即负值），这就是为什么前面的代码在计算平方根之前先计算 `-scores`。

来看下结果：

```
>>> def display_scores(scores):
...     print("Scores:", scores)
...     print("Mean:", scores.mean())
...     print("Standard deviation:", scores.std())
...
>>> display_scores(tree_rmse_scores)
Scores: [ 74678.4916885   64766.2398337   69632.86942005  69166.67693232
          71486.76507766  73321.65695983  71860.04741226  71086.32691692
          76934.2726093   69060.93319262]
Mean: 71199.4280043
Standard deviation: 3202.70522793
```

现在决策树就不像前面看起来那么好了。实际上，它看起来比线性回归模型还糟！注意到交叉验证不仅可以让你得到模型性能的评估，还能测量评估的准确性（即，它的标准差）。决策树的评分大约是 71200，通常波动有 ±3200。如果只有一个验证集，就得不到这些信息。但是交叉验证的代价是训练了模型多次，不可能总是这样。

让我们计算下线性回归模型的的相同分数，以做确保：

```
>>> lin_scores = cross_val_score(lin_reg, housing_prepared, housing_labels,
...                              scoring="neg_mean_squared_error", cv=10)
...
>>> lin_rmse_scores = np.sqrt(-lin_scores)
>>> display_scores(lin_rmse_scores)
Scores: [ 70423.5893262   65804.84913139  66620.84314068  72510.11362141
          66414.74423281  71958.89083606  67624.90198297  67825.36117664
          72512.36533141  68028.11688067]
Mean: 68972.377566
Standard deviation: 2493.98819069
```

判断没错：决策树模型过拟合很严重，它的性能比线性回归模型还差。

现在再尝试最后一个模型：`RandomForestRegressor`。第7章我们会看到，随机森林是通过用特征的随机子集训练许多决策树。在其它多个模型之上建立模型称为集成学习（Ensemble Learning），它是推进 ML 算法的一种好方法。我们会跳过大部分的代码，因为代码本质上和其它模型一样：

```
>>> from sklearn.ensemble import RandomForestRegressor
>>> forest_reg = RandomForestRegressor()
>>> forest_reg.fit(housing_prepared, housing_labels)
>>> [...]
>>> forest_rmse
22542.396440343684
>>> display_scores(forest_rmse_scores)
Scores: [ 53789.2879722   50256.19806622  52521.55342602  53237.44937943
          52428.82176158  55854.61222549  52158.02291609  50093.66125649
          53240.80406125  52761.50852822]
Mean: 52634.1919593
Standard deviation: 1576.20472269
```

现在好多了：随机森林看起来很有希望。但是，训练集的评分仍然比验证集的评分低很多。解决过拟合可以通过简化模型，给模型加限制（即，规整化），或用更多的训练数据。在深入随机森林之前，你应该尝试下机器学习算法的其它类型模型（不同核心的支持向量机，神经网络，等等），不要在调节超参数上花费太多时间。目标是列出一个可能模型的列表（两到五个）。

> 提示：你要保存每个试验过的模型，以便后续可以再用。要确保有超参数和训练参数，以及交叉验证评分，和实际的预测值。这可以让你比较不同类型模型的评分，还可以比较误差种类。你可以用 Python 的模块`pickle`，非常方便地保存 Scikit-Learn 模型，或使用`sklearn.externals.joblib`，后者序列化大 NumPy 数组更有效率：
>
> ```
> from sklearn.externals import joblib
>
> joblib.dump(my_model, "my_model.pkl")
> # 然后
> my_model_loaded = joblib.load("my_model.pkl")
> ```

模型微调

假设你现在有了一个列表，列表里有几个有希望的模型。你现在需要对它们进行微调。让我们来看几种微调的方法。

网格搜索

微调的一种方法是手工调整超参数，直到找到一个好的超参数组合。这么做的话会非常冗长，你也可能没有时间探索多种组合。

你应该使用 Scikit-Learn 的 `GridSearchCV` 来做这项搜索工作。你所需要做的是告诉 `GridSearchCV` 要试验有哪些超参数，要试验什么值，`GridSearchCV` 就能用交叉验证试验所有可能超参数值的组合。例如，下面的代码搜索了 `RandomForestRegressor` 超参数值的最佳组合：

```
from sklearn.model_selection import GridSearchCV

param_grid = [
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
  ]

forest_reg = RandomForestRegressor()

grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error')

grid_search.fit(housing_prepared, housing_labels)
```

> 当你不能确定超参数该有什么值，一个简单的方法是尝试连续的 10 的幂（如果想要一个粒度更小的搜寻，可以用更小的数，就像在这个例子中对超参数 `n_estimators` 做的）。

`param_grid` 告诉 Scikit-Learn 首先评估所有的列在第一个 `dict` 中的 `n_estimators` 和 `max_features` 的 `3 × 4 = 12` 种组合（不用担心这些超参数的含义，会在第 7 章中解释）。然后尝试第二个 `dict` 中超参数的 `2 × 3 = 6` 种组合，这次会将超参数 `bootstrap` 设为 `False` 而不是 `True`（后者是该超参数的默认值）。

总之，网格搜索会探索 `12 + 6 = 18` 种 `RandomForestRegressor` 的超参数组合，会训练每个模型五次（因为用的是五折交叉验证）。换句话说，训练总共有 `18 × 5 = 90` 轮！K 折将要花费大量时间，完成后，你就能获得参数的最佳组合，如下所示：

```
>>> grid_search.best_params_
{'max_features': 6, 'n_estimators': 30}
```

> 提示：因为 30 是 `n_estimators` 的最大值，你也应该估计更高的值，因为评估的分数可能会随 `n_estimators` 的增大而持续提升。

你还能直接得到最佳的估计器：

```
>>> grid_search.best_estimator_
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
           max_features=6, max_leaf_nodes=None, min_samples_leaf=1,
           min_samples_split=2, min_weight_fraction_leaf=0.0,
           n_estimators=30, n_jobs=1, oob_score=False, random_state=None,
           verbose=0, warm_start=False)
```

> 注意：如果 `GridSearchCV` 是以（默认值） `refit=True` 开始运行的，则一旦用交叉验证找到了最佳的估计器，就会在整个训练集上重新训练。这是一个好方法，因为用更多数据训练会提高性能。

当然，也可以得到评估得分：

```
>>> cvres = grid_search.cv_results_
... for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
...     print(np.sqrt(-mean_score), params)
...
64912.0351358 {'max_features': 2, 'n_estimators': 3}
55535.2786524 {'max_features': 2, 'n_estimators': 10}
52940.2696165 {'max_features': 2, 'n_estimators': 30}
60384.0908354 {'max_features': 4, 'n_estimators': 3}
52709.9199934 {'max_features': 4, 'n_estimators': 10}
50503.5985321 {'max_features': 4, 'n_estimators': 30}
59058.1153485 {'max_features': 6, 'n_estimators': 3}
52172.0292957 {'max_features': 6, 'n_estimators': 10}
49958.9555932 {'max_features': 6, 'n_estimators': 30}
59122.260006 {'max_features': 8, 'n_estimators': 3}
52441.5896087 {'max_features': 8, 'n_estimators': 10}
50041.4899416 {'max_features': 8, 'n_estimators': 30}
62371.1221202 {'bootstrap': False, 'max_features': 2, 'n_estimators': 3}
54572.2557534 {'bootstrap': False, 'max_features': 2, 'n_estimators': 10}
59634.0533132 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
52456.0883904 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
58825.665239 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
52012.9945396 {'bootstrap': False, 'max_features': 4, 'n_estimators': 10}
```

在这个例子中，我们通过设定超参数 `max_features` 为 6， `n_estimators` 为 30，得到了最佳方案。对这个组合，RMSE 的值是 49959，这比之前使用默认的超参数的值（52634）要稍微好一些。祝贺你，你成功地微调了最佳模型！

> 提示：不要忘记，你可以像超参数一样处理数据准备的步骤。例如，网格搜索可以自动判断是否添加一个你不确定的特征（比如，使用转换器 `CombinedAttributesAdder` 的超参数 `add_bedrooms_per_room` ）。它还能用相似的方法来自动找到处理异常值、缺失特征、特征选择等任务的最佳方法。

随机搜索

当探索相对较少的组合时，就像前面的例子，网格搜索还可以。但是当超参数的搜索空间很大时，最好使用 `RandomizedSearchCV` 。这个类的使用方法和类 `GridSearchCV` 很相似，但它不是尝试所有可能的组合，而是通过选择每个超参数的一个随机值的特定数量的随机组合。这个方法有两个优点：

- 如果你让随机搜索运行，比如 1000 次，它会探索每个超参数的 1000 个不同的值（而不是像网格搜索那样，只搜索每个超参数的几个值）。
- 你可以方便地通过设定搜索次数，控制超参数搜索的计算量。

集成方法

另一种微调系统的方法是将表现最好的模型组合起来。组合（集成）之后的性能通常要比单独的模型要好（就像随机森林要比单独的决策树要好），特别是当单独模型的误差类型不同时。我们会在第7章更深入地讲解这点。

分析最佳模型和它们的误差

通过分析最佳模型，常常可以获得对问题更深的了解。比如，`RandomForestRegressor` 可以指出每个属性对于做出准确预测的相对重要性：

```
>>> feature_importances = grid_search.best_estimator_.feature_importances_
>>> feature_importances
array([  7.14156423e-02,   6.76139189e-02,   4.44260894e-02,
         1.66308583e-02,   1.66076861e-02,   1.82402545e-02,
         1.63458761e-02,   3.26497987e-01,   6.04365775e-02,
         1.13055290e-01,   7.79324766e-02,   1.12166442e-02,
         1.53344918e-01,   8.41308969e-05,   2.68483884e-03,
         3.46681181e-03])
```

将重要性分数和属性名放到一起：

```
>>> extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
>>> cat_one_hot_attribs = list(encoder.classes_)
>>> attributes = num_attribs + extra_attribs + cat_one_hot_attribs
>>> sorted(zip(feature_importances,attributes), reverse=True)
[(0.32649798665134971, 'median_income'),
 (0.15334491760305854, 'INLAND'),
 (0.11305529021187399, 'pop_per_hhold'),
 (0.07793247662544775, 'bedrooms_per_room'),
 (0.071415642259275158, 'longitude'),
 (0.067613918945568688, 'latitude'),
 (0.060436577499703222, 'rooms_per_hhold'),
 (0.04442608939578685, 'housing_median_age'),
 (0.018240254462909437, 'population'),
 (0.01663085833886218, 'total_rooms'),
 (0.016607686091288865, 'total_bedrooms'),
 (0.016345876147580776, 'households'),
 (0.011216644219017424, '<1H OCEAN'),
 (0.0034668118081117387, 'NEAR OCEAN'),
 (0.0026848388432755429, 'NEAR BAY'),
 (8.4130896890070617e-05, 'ISLAND')]
```

有了这个信息，你就可以丢弃一些不那么重要的特征（比如，显然只要一个 `ocean_proximity` 的类型（INLAND）就够了，所以可以丢弃掉其它的）。

你还应该看一下系统犯的误差，搞清为什么会有些误差，以及如何改正问题（添加更多的特征，或相反，去掉没有什么信息的特征，清洗异常值等等）。

用测试集评估系统

调节完系统之后，你终于有了一个性能足够好的系统。现在就可以用测试集评估最后的模型了。这个过程没有什么特殊的：从测试集得到预测值和标签，运行 `full_pipeline` 转换数据（调用 `transform()`，而不是 `fit_transform()`！），再用测试集评估最终模型：

```
final_model = grid_search.best_estimator_

X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].copy()

X_test_prepared = full_pipeline.transform(X_test)

final_predictions = final_model.predict(X_test_prepared)

final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse)   # => evaluates to 48,209.6
```

评估结果通常要比交叉验证的效果差一点，如果你之前做过很多超参数微调（因为你的系统在验证集上微调，得到了不错的性能，通常不会在未知的数据集上有同样好的效果）。这个例子不属于这种情况，但是当发生这种情况时，你一定要忍住不要调节超参数，使测试集的效果变好；这样的提升不能推广到新数据上。

然后就是项目的预上线阶段：你需要展示你的方案（重点说明学到了什么、做了什么、没做什么、做过什么假设、系统的限制是什么，等等），记录下所有事情，用漂亮的图表和容易记住的表达（比如，“收入中位数是房价最重要的预测量”）做一次精彩的展示。

启动、监控、维护系统

很好，你被允许启动系统了！你需要为实际生产做好准备，特别是接入输入数据源，并编写测试。

你还需要编写监控代码，以固定间隔检测系统的实时表现，当发生下降时触发报警。这对于捕获突然的系统崩溃和性能下降十分重要。做监控很常见，是因为模型会随着数据的演化而性能下降，除非模型用新数据定期训练。

评估系统的表现需要对预测值采样并进行评估。这通常需要人来分析。分析者可能是领域专家，或者是众包平台（比如 Amazon Mechanical Turk 或 CrowdFlower）的工人。不管采用哪种方法，你都需要将人工评估的流水线植入系统。

你还要评估系统输入数据的质量。有时因为低质量的信号（比如失灵的传感器发送随机值，或另一个团队的输出停滞），系统的表现会逐渐变差，但可能需要一段时间，系统的表现才能下降到一定程度，触发警报。如果监测了系统的输入，你就可能尽量早的发现问题。对于线上学习系统，监测输入数据是非常重要的。

最后，你可能想定期用新数据训练模型。你应该尽可能自动化这个过程。如果不这么做，非常有可能你需要每隔至少六个月更新模型，系统的表现就会产生严重波动。如果你的系统是一个线上学习系统，你需要定期保存系统状态快照，好能方便地回滚到之前的工作状态。

实践！

希望这一章能告诉你机器学习项目是什么样的，你能用学到的工具训练一个好系统。你已经看到，大部分的工作是数据准备步骤、搭建监测工具、建立人为评估的流水线和自动化定期模型训练，当然，最好能了解整个过程、熟悉三或四种算法，而不是在探索高级算法上浪费全部时间，导致在全局上的时间不够。

因此，如果你还没这样做，现在最好拿起台电脑，选择一个感兴趣的数据集，将整个流程从头到尾完成一遍。一个不错的着手开始的地点是竞赛网站，比如 http://kaggle.com/：你会得到一个数据集，一个目标，以及分享经验的人。

练习

使用本章的房产数据集：

1. 尝试一个支持向量机回归器（`sklearn.svm.SVR`），使用多个超参数，比如 `kernel="linear"`（多个超参数 `C` 值）。现在不用担心这些超参数是什么含义。最佳的 `SVR` 预测表现如何？
2. 尝试用 `RandomizedSearchCV` 替换 `GridSearchCV`。
3. 尝试在准备流水线中添加一个只选择最重要属性的转换器。
4. 尝试创建一个单独的可以完成数据准备和最终预测的流水线。
5. 使用 `GridSearchCV` 自动探索一些准备过程中的候选项。

练习题答案可以在线上的 Jupyter notebook 找到。

第3章 分类

在第一章我们提到过最常用的监督学习任务是回归（用于预测某个值）和分类（预测某个类别）。在第二章我们探索了一个回归任务：预测房价。我们使用了多种算法，诸如线性回归，决策树，和随机森林（这个将会在后面的章节更详细地讨论）。现在我们将我们的注意力转到分类任务上。

MNIST

在本章当中，我们将会使用 MNIST 这个数据集，它有着 70000 张规格较小的手写数字图片，由美国的高中生和美国人口调查局的职员手写而成。这相当于机器学习当中的“Hello World”，人们无论什么时候提出一个新的分类算法，都想知道该算法在这个数据集上的表现如何。机器学习的初学者迟早也会处理 MNIST 这个数据集。

Scikit-Learn 提供了许多辅助函数，以便于下载流行的数据集。MNIST 是其中一个。下面的代码获取 MNIST

```
>>> from sklearn.datasets import fetch_mldata
>>> mnist = fetch_mldata('MNIST original')
>>> mnist
{'COL_NAMES': ['label', 'data'],
'DESCR': 'mldata.org dataset: mnist-original',
'data': array([[0, 0, 0, ..., 0, 0, 0],
                [0, 0, 0, ..., 0, 0, 0],
                [0, 0, 0, ..., 0, 0, 0],
                ...,
                [0, 0, 0, ..., 0, 0, 0],
                [0, 0, 0, ..., 0, 0, 0],
                [0, 0, 0, ..., 0, 0, 0]], dtype=uint8),
'target': array([ 0., 0., 0., ..., 9., 9., 9.])}
```

一般而言，由 sklearn 加载的数据集有着相似的字典结构，这包括：

- `DESCR` 键描述数据集
- `data` 键存放一个数组，数组的一行表示一个样例，一列表示一个特征
- `target` 键存放一个标签数组

让我们看一下这些数组

```
>>> X, y = mnist["data"], mnist["target"]
>>> X.shape
(70000, 784)
>>> y.shape
(70000,)
```

MNIST 有 70000 张图片，每张图片有 784 个特征。这是因为每个图片都是 `28*28` 像素的，并且每个像素的值介于 0~255 之间。让我们看一看数据集的某一个数字。你只需要将某个实例的特征向量， `reshape` 为 `28*28` 的数组，然后使用 Matplotlib 的 `imshow` 函数展示出来。

```
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
some_digit = X[36000]
some_digit_image = some_digit.reshape(28, 28)
plt.imshow(some_digit_image, cmap = matplotlib.cm.binary, interpolation="nearest")
plt.axis("off")
plt.show()
```



这看起来像个 5，实际上它的标签告诉我们：

```
>>> y[36000]
5.0
```

图3-1 展示了一些来自 MNIST 数据集的图片。当你处理更加复杂的分类任务的时候，它会让你更有感觉。



先等一下！你总是应该先创建测试集，并且在验证数据之前先把测试集晾到一边。MNIST 数据集已经事先被分成了一个训练集（前 60000 张图片）和一个测试集（最后 10000 张图片）

```
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

让我们打乱训练集。这可以保证交叉验证的每一折都是相似（你不会期待某一折缺少某类数字）。而且，一些学习算法对训练样例的顺序敏感，当它们在一行当中得到许多相似的样例，这些算法将会表现得非常差。打乱数据集将保证这种情况不会发生。

```
import numpy as np

shuffle_index = np.random.permutation(60000)
X_train, y_train = X_train[shuffle_index], y_train[shuffle_index]
```

训练一个二分类器

现在我们简化一下问题，只尝试去识别一个数字，比如说，数字 5。这个“数字 5 检测器”就是一个二分类器，能够识别两类别，“是 5”和“非 5”。让我们为这个分类任务创建目标向量：

```
y_train_5 = (y_train == 5) # True for all 5s, False for all other digits.
y_test_5 = (y_test == 5)
```

现在让我们挑选一个分类器去训练它。用随机梯度下降分类器 SGD，是一个不错的开始。使用 Scikit-Learn 的 `SGDClassifier` 类。这个分类器有一个好处是能够高效地处理非常大的数据集。这部分原因在于SGD一次只处理一条数据，这也使得 SGD 适合在线学习（online learning）。我们在稍后会看到它。让我们创建一个 `SGDClassifier` 和在整个数据集上训练它。

```
from sklearn.linear_model import SGDClassifier
sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train_5)
```

> `SGDClassifier` 依赖于训练集的随机程度（所以被命名为 stochastic，随机之义）。如果你想重现结果，你应该固定参数 `random_state`

现在你可以用它来查出数字 5 的图片。

```
>>> sgd_clf.predict([some_digit])
array([ True], dtype=bool)
```

分类器猜测这个数字代表 5（ `True` ）。看起来在这个例子当中，它猜对了。现在让我们评估这个模型的性能。

对性能的评估

评估一个分类器，通常比评估一个回归器更加玄学。所以我们将会花大量的篇幅在这个话题上。有许多量度性能的方法，所以拿来一杯咖啡和准备学习许多新概念和首字母缩略词吧。

使用交叉验证测量准确性

评估一个模型的好方法是使用交叉验证，就像第二章所做的那样。

> **实现交叉验证**

在交叉验证过程中，有时候你会需要更多的控制权，相较于函数 `cross_val_score()` 或者其他相似函数所提供的功能。这种情况下，你可以实现你自己版本的交叉验证。事实上它相当直接。以下代码粗略地做了和 `cross_val_score()` 相同的事情，并且输出相同的结果。

```
from sklearn.model_selection import StratifiedKFold
from sklearn.base import clone
skfolds = StratifiedKFold(n_splits=3, random_state=42)
for train_index, test_index in skfolds.split(X_train, y_train_5):
    clone_clf = clone(sgd_clf)
    X_train_folds = X_train[train_index]
    y_train_folds = (y_train_5[train_index])
    X_test_fold = X_train[test_index]
    y_test_fold = (y_train_5[test_index])
    clone_clf.fit(X_train_folds, y_train_folds)
    y_pred = clone_clf.predict(X_test_fold)
    n_correct = sum(y_pred == y_test_fold)
    print(n_correct / len(y_pred)) # prints 0.9502, 0.96565 and 0.96495
```

> `StratifiedKFold` 类实现了分层采样（详见第二章的解释），生成的折（fold）包含了各类相应比例的样例。在每一次迭代，上述代码生成分类器的一个克隆版本，在训练折（training folds）的克隆版本上进行训，在测试折（test folds）上进行预测。然后它计算出被正确预测的数目和输出正确预测的比例。

让我们使用 `cross_val_score()` 函数来评估 `SGDClassifier` 模型，同时使用 K 折交叉验证，此处让 `k=3` 。记住：K 折交叉验证意味着把训练集分成 K 折（此处 3 折），然后使用一个模型对其中一折进行预测，对其他折进行训练。

```
>>> from sklearn.model_selection import cross_val_score
>>> cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([ 0.9502 , 0.96565, 0.96495]
```

哇！在交叉验证上有大于 95% 的精度（accuracy）？这看起来很令人吃惊。先别高兴，让我们来看一个非常笨的分类器去分类，看看其在“非 5”这个类上的表现。

```
from sklearn.base import BaseEstimator
class Never5Classifier(BaseEstimator):
    def fit(self, X, y=None):
        pass
    def predict(self, X):
        return np.zeros((len(X), 1), dtype=bool)
```

你能猜到这个模型的精度吗？揭晓谜底：

```
>>> never_5_clf = Never5Classifier()
>>> cross_val_score(never_5_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([ 0.909 , 0.90715, 0.9128 ])
```

没错，这个笨的分类器也有 90% 的精度。这是因为只有 10% 的图片是数字 5，所以你总是猜测某张图片不是 5，你也会有90%的可能性是对的。

这证明了为什么精度通常来说不是一个好的性能度量指标，特别是当你处理有偏差的数据集，比方说其中一些类比其他类频繁得多。

混淆矩阵

对分类器来说，一个好得多的性能评估指标是混淆矩阵。大体思路是：输出类别A被分类成类别 B 的次数。举个例子，为了知道分类器将 5 误分为 3 的次数，你需要查看混淆矩阵的第五航第三列。

为了计算混淆矩阵，首先你需要有一系列的预测值，这样才能将预测值与真实值做比较。你或许想在测试集上做预测。但是我们现在先不碰它。（记住，只有当你处于项目的尾声，当你准备上线一个分类器的时候，你才应该使用测试集）。相反，你应该使用 `cross_val_predict()` 函数

```
from sklearn.model_selection import cross_val_predict
y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```

就像 `cross_val_score()` ， `cross_val_predict()` 也使用 K 折交叉验证。它不是返回一个评估分数，而是返回基于每一个测试折做出的一个预测值。这意味着，对于每一个训练集的样例，你得到一个干净的预测（“干净”是说一个模型在训练过程当中没有用到测试集的数据）。

现在使用 `confusion_matrix()` 函数，你将会得到一个混淆矩阵。传递目标类(`y_train_5`)和预测类（ `y_train_pred` ）给它。

```
>>> from sklearn.metrics import confusion_matrix
>>> confusion_matrix(y_train_5, y_train_pred)
array([[53272, 1307],
        [ 1077, 4344]])
```

混淆矩阵中的每一行表示一个实际的类, 而每一列表示一个预测的类。该矩阵的第一行认为“非 5”（反例）中的 53272 张被正确归类为 “非 5”（他们被称为真反例，true negatives）, 而其余 1307 被错误归类为"是 5" （假正例，false positives）。第二行认为“是 5” （正例）中的 1077 被错误地归类为“非 5”（假反例，false negatives），其余 4344 正确分类为 “是 5”类（真正例，true positives）。一个完美的分类器将只有真反例和真正例，所以混淆矩阵的非零值仅在其主对角线（左上至右下）。

```
>>> confusion_matrix(y_train_5, y_train_perfect_predictions)
array([[54579, 0],
        [ 0, 5421]])
```

混淆矩阵可以提供很多信息。有时候你会想要更加简明的指标。一个有趣的指标是正例预测的精度，也叫做分类器的准确率（precision）。

公式 3-1 准确率

$$precision = \frac{TP}{TP + FP}$$

其中 TP 是真正例的数目，FP 是假正例的数目。

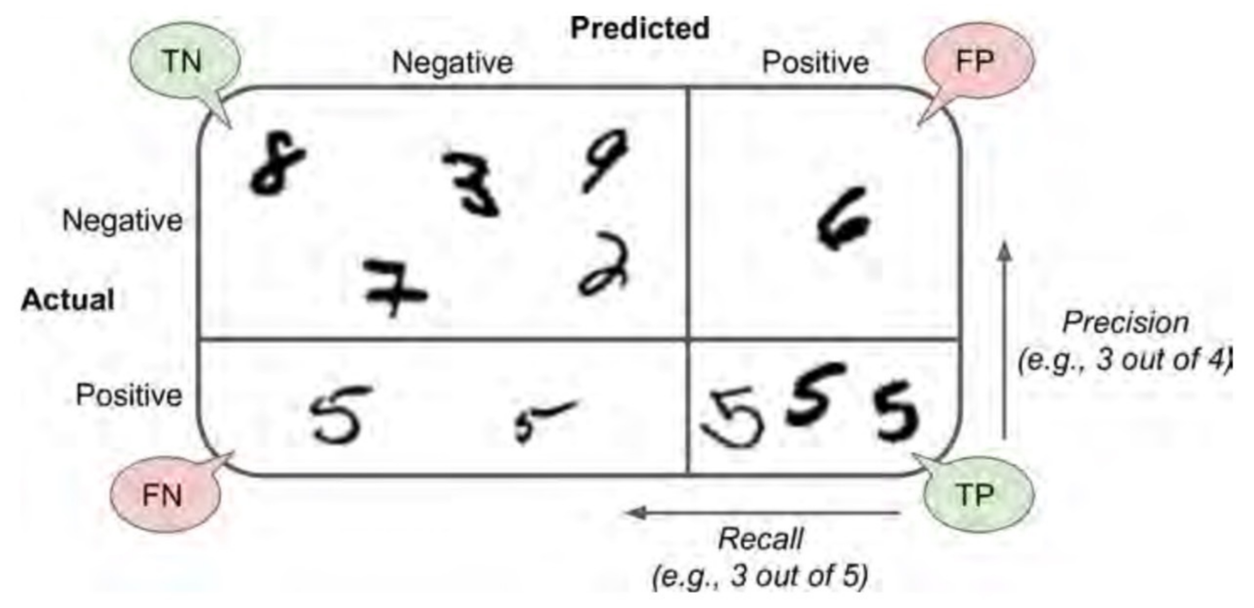
想要一个完美的准确率，一个平凡的方法是构造一个单一正例的预测和确保这个预测是正确的（`precision = 1/1 = 100%`）。但是这什么用，因为分类器会忽略所有样例，除了那一个正例。所以准确率一般会伴随另一个指标一起使用，这个指标叫做召回率（recall），也叫做敏感度（sensitivity）或者真正例率（true positive rate， TPR）。这是正例被分类器正确探测出的比率。

公式 3-2 Recall

$$recall = \frac{TP}{TP + FN}$$

FN 是假反例的数目。

如果你对于混淆矩阵感到困惑，图 3-2 将对你有帮助



准确率与召回率

Scikit-Learn 提供了一些函数去计算分类器的指标，包括准确率和召回率。

```
>>> from sklearn.metrics import precision_score, recall_score
>>> precision_score(y_train_5, y_pred) # == 4344 / (4344 + 1307)
0.76871350203503808
>>> recall_score(y_train_5, y_train_pred) # == 4344 / (4344 + 1077)
0.79136690647482011
```

当你去观察精度的时候，你的“数字 5 探测器”看起来还不够好。当它声明某张图片是 5 的时候，它只有 77% 的可能性是正确的。而且，它也只检测出“是 5”类图片当中的 79%。

通常结合准确率和召回率会更加方便，这个指标叫做“F1 值”，特别是当你需要一个简单的方法去比较两个分类器的优劣的时候。F1 值是准确率和召回率的调和平均。普通的平均值平等地看待所有的值，而调和平均会给小的值更大的权重。所以，要想分类器得到一个高的 F1 值，需要召回率和准确率同时高。

公式 3-3 F1 值

$$F1 = \frac{2}{\frac{1}{precision} + \frac{1}{recall}} = 2 * \frac{precison * recall}{precison + recall} = \frac{TP}{TP + \frac{FN+FP}{2}}$$

为了计算 F1 值，简单调用 `f1_score()`

```
>>> from sklearn.metrics import f1_score
>>> f1_score(y_train_5, y_pred)
0.78468208092485547
```

F1 支持那些有着相近准确率和召回率的分类器。这不会总是你想要的。有的场景你会绝大程度地关心准确率，而另外一些场景你会更关心召回率。举例子，如果你训练一个分类器去检测视频是否适合儿童观看，你会倾向选择那种即便拒绝了很多好视频、但保证所保留的视频都是好（高准确率）的分类器，而不是那种高召回率、但让坏视频混入的分类器（这种情况下你或许想增加人工去检测分类器选择出来的视频）。另一方面，加入你训练一个分类器去检测监控图像当中的窃贼，有着 30% 准确率、99% 召回率的分类器或许是合适的（当然，警卫会得到一些错误的报警，但是几乎所有的窃贼都会被抓到）。

不幸的是，你不能同时拥有两者。增加准确率会降低召回率，反之亦然。这叫做准确率与召回率之间的折衷。

准确率/召回率之间的折衷

为了弄懂这个折衷，我们看一下 `SGDClassifier` 是如何做分类决策的。对于每个样例，它根据决策函数计算分数,如果这个分数大于一个阈值，它会将样例分配给正例，否则它将分配给反例。图 3-3 显示了几个数字从左边的最低分数排到右边的最高分。假设决策阈值位于中间的箭头（介于两个 5 之间）：您将发现4个真正例（数字 5）和一个假正例（数字 6）在该阈值的右侧。因此,使用该阈值,准确率为 80%（4/5）。但实际有 6 个数字 5，分类器只检测 4 个,所以召回是 67% （4/6）。现在，如果你 提高阈值（移动到右侧的箭头），假正例（数字 6）成为一个真反例，从而提高准确率（在这种情况下高达 100%），但一个真正例 变成假反例，召回率降低到 50%。相反，降低阈值可提高召回率、降低准确率。

![图3-3 决策阈值与准确度/召回率折衷][../images/chapter_3/chapter3.3.jpeg]

Scikit-Learn 不让你直接设置阈值，但是它给你提供了设置决策分数的方法，这个决策分数可以用来产生预测。它不是调用分类器的 `predict()` 方法，而是调用 `decision_function()` 方法。这个方法返回每一个样例的分数值，然后基于这个分数值，使用你想要的任何阈值做出预测。

```
>>> y_scores = sgd_clf.decision_function([some_digit])
>>> y_scores
array([ 161855.74572176])
>>> threshold = 0
>>> y_some_digit_pred = (y_scores > threshold)
array([ True], dtype=bool)
```

`SGDClassifier` 用了一个等于 0 的阈值，所以前面的代码返回了跟 `predict()` 方法一样的结果（都返回了 `true` ）。让我们提高这个阈值：

```
>>> threshold = 200000
>>> y_some_digit_pred = (y_scores > threshold)
>>> y_some_digit_pred
array([False], dtype=bool)
```

这证明了提高阈值会降调召回率。这个图片实际就是数字 5，当阈值等于 0 的时候，分类器可以探测到这是一个 5，当阈值提高到 20000 的时候，分类器将不能探测到这是数字 5。

那么，你应该如何使用哪个阈值呢？首先，你需要再次使用 `cross_val_predict()` 得到每一个样例的分数值，但是这一次指定返回一个决策分数，而不是预测值。

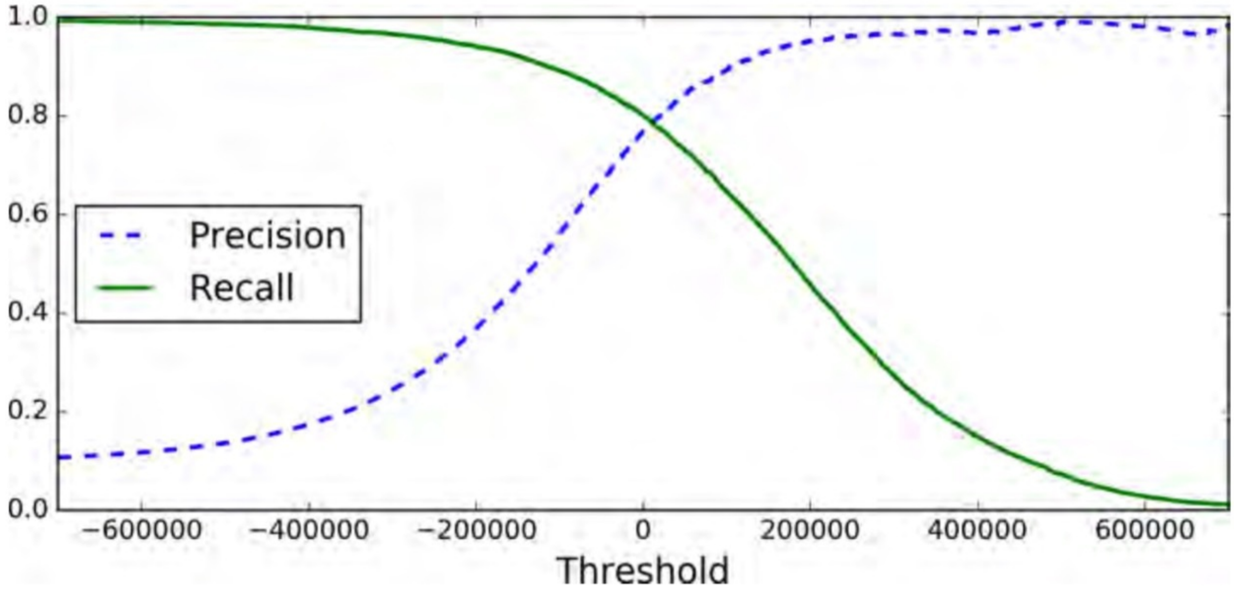
```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                             method="decision_function")
```

现在有了这些分数值。对于任何可能的阈值，使用 `precision_recall_curve()` ,你都可以计算准确率和召回率:

```
from sklearn.metrics import precision_recall_curve
precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
```

最后，你可以使用 Matplotlib 画出准确率和召回率（图 3-4），这里把准确率和召回率当作是阈值的一个函数。

```
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision")
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall")
    plt.xlabel("Threshold")
    plt.legend(loc="upper left")
    plt.ylim([0, 1])
plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
plt.show()
```



> 你也许会好奇为什么准确率曲线比召回率曲线更加起伏不平。原因是准确率有时候会降低，尽管当你提高阈值的时候，通常来说准确率会随之提高。回头看图 3-3，留意当你从中间箭头开始然后向右移动一个数字会发生什么： 准确率会由 4/5（80%）降到 3/4（75%）。另一方面，当阈值提高时候，召回率只会降低。这也就说明了为什么召回率的曲线更加平滑。

现在你可以选择适合你任务的最佳阈值。另一个选出好的准确率/召回率折衷的方法是直接画出准确率对召回率的曲线，如图 3-5 所示。

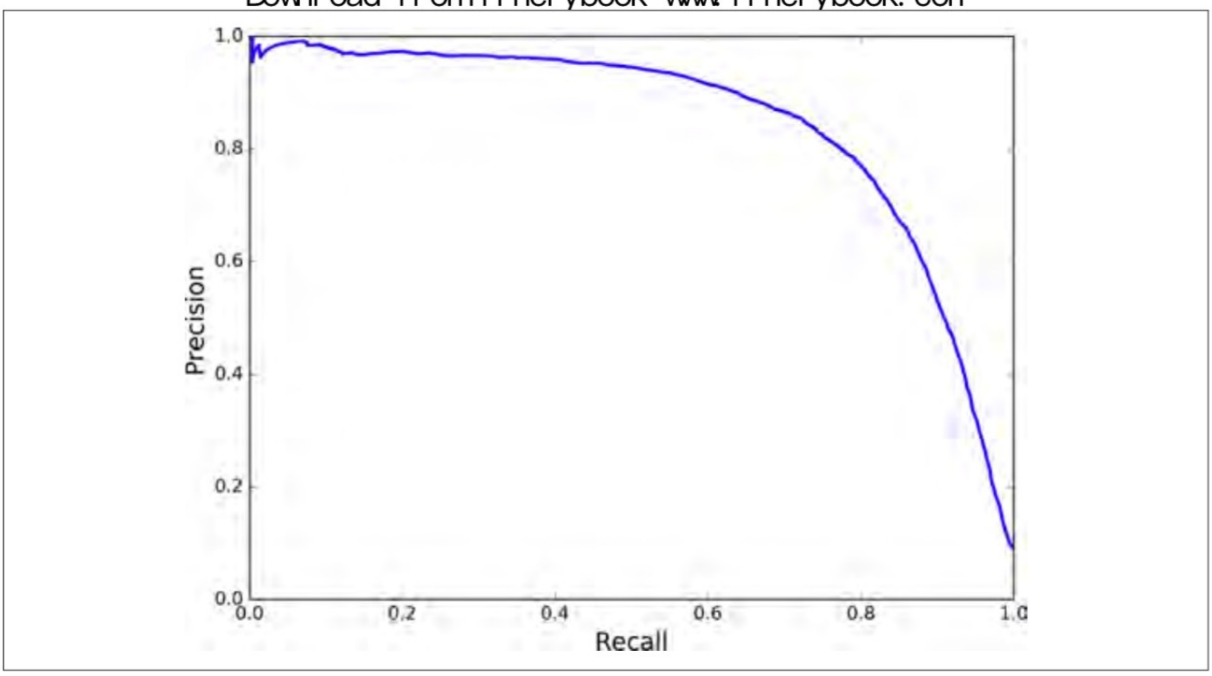


Figure 3-5. Precision versus recall

可以看到，在召回率在 80% 左右的时候，准确率急剧下降。你可能会想选择在急剧下降之前选择出一个准确率/召回率折衷点。比如说，在召回率 60% 左右的点。当然，这取决于你的项目需求。

我们假设你决定达到 90% 的准确率。你查阅第一幅图（放大一些），在 70000 附近找到一个阈值。为了作出预测（目前为止只在训练集上预测），你可以运行以下代码，而不是运行分类器的 `predict()` 方法。

```
y_train_pred_90 = (y_scores > 70000)
```

让我们检查这些预测的准确率和召回率：

```
>>> precision_score(y_train_5, y_train_pred_90)
0.8998702983138781
>>> recall_score(y_train_5, y_train_pred_90)
0.63991883416343853
```

很棒！你拥有了一个（近似） 90% 准确率的分类器。它相当容易去创建一个任意准确率的分类器，只要将阈值设置得足够高。但是，一个高准确率的分类器不是非常有用，如果它的召回率太低！

> 如果有人说“让我们达到 99% 的准确率”，你应该问“相应的召回率是多少？”

ROC 曲线

受试者工作特征（ROC）曲线是另一个二分类器常用的工具。它非常类似与准确率/召回率曲线，但不是画出准确率对召回率的曲线，ROC 曲线是真正例率（true positive rate，另一个名字叫做召回率）对假正例率（false positive rate, FPR）的曲线。FPR 是反例被错误分成正例的比率。它等于 1 减去真反例率（true negative rate， TNR）。TNR是反例被正确分类的比率。TNR也叫做特异性。所以 ROC 曲线画出召回率对（1 减特异性）的曲线。

为了画出 ROC 曲线，你首先需要计算各种不同阈值下的 TPR、FPR，使用 `roc_curve()` 函数：

```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
```

然后你可以使用 matplotlib，画出 FPR 对 TPR 的曲线。下面的代码生成图 3-6.

```
def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.axis([0, 1, 0, 1])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
plot_roc_curve(fpr, tpr)
plt.show()
```

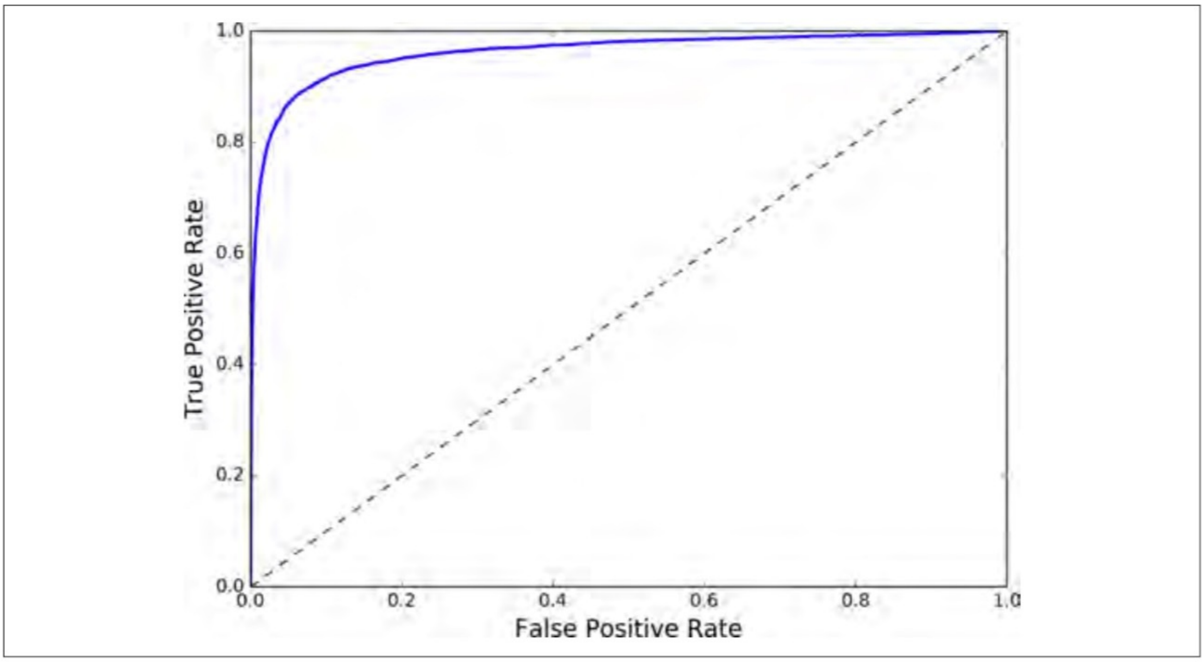


Figure 3-6. ROC curve

这里同样存在折衷的问题：召回率（TPR）越高，分类器就会产生越多的假正例（FPR）。图中的点线是一个完全随机的分类器生成的 ROC 曲线；一个好的分类器的 ROC 曲线应该尽可能远离这条线（即向左上角方向靠拢）。

一个比较分类器之间优劣的方法是：测量ROC曲线下的面积（AUC）。一个完美的分类器的 ROC AUC 等于 1，而一个纯随机分类器的 ROC AUC 等于 0.5。Scikit-Learn 提供了一个函数来计算 ROC AUC：

```
>>> from sklearn.metrics import roc_auc_score
>>> roc_auc_score(y_train_5, y_scores)
0.97061072797174941
```

因为 ROC 曲线跟准确率/召回率曲线（或者叫 PR）很类似，你或许会好奇如何决定使用哪一个曲线呢？一个笨拙的规则是，优先使用 PR 曲线当正例很少，或者当你关注假正例多于假反例的时候。其他情况使用 ROC 曲线。举例子，回顾前面的 ROC 曲线和 ROC AUC 数值，你或许人为这个分类器很棒。但是这几乎全是因为只有少数正例（“是 5”），而大部分是反例（“非 5”）。相反，PR 曲线清楚显示出这个分类器还有很大的改善空间（PR 曲线应该尽可能地靠近右上角）。

让我们训练一个 `RandomForestClassifier` ，然后拿它的的ROC曲线和ROC AUC数值去跟 `SGDClassifier` 的比较。首先你需要得到训练集每个样例的数值。但是由于随机森林分类器的工作方式， `RandomForestClassifier` 不提供 `decision_function()` 方法。相反，它提供了 `predict_proba()` 方法。Skikit-Learn分类器通常二者中的一个。 `predict_proba()` 方法返回一个数组，数组的每一行代表一个样例，每一列代表一个类。数组当中的值的意思是：给定一个样例属于给定类的概率。比如，70%的概率这幅图是数字 5。

```
from sklearn.ensemble import RandomForestClassifier
forest_clf = RandomForestClassifier(random_state=42)
y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3,
                                    method="predict_proba")
```

但是要画 ROC 曲线，你需要的是样例的分数，而不是概率。一个简单的解决方法是使用正例的概率当作样例的分数。

```
y_scores_forest = y_probas_forest[:, 1] # score = proba of positive class
fpr_forest, tpr_forest, thresholds_forest = roc_curve(y_train_5,y_scores_forest)
```

现在你即将得到 ROC 曲线。将前面一个分类器的 ROC 曲线一并画出来是很有用的，可以清楚地进行比较。见图 3-7。

```
plt.plot(fpr, tpr, "b:", label="SGD")
plot_roc_curve(fpr_forest, tpr_forest, "Random Forest")
plt.legend(loc="bottom right")
plt.show()
```

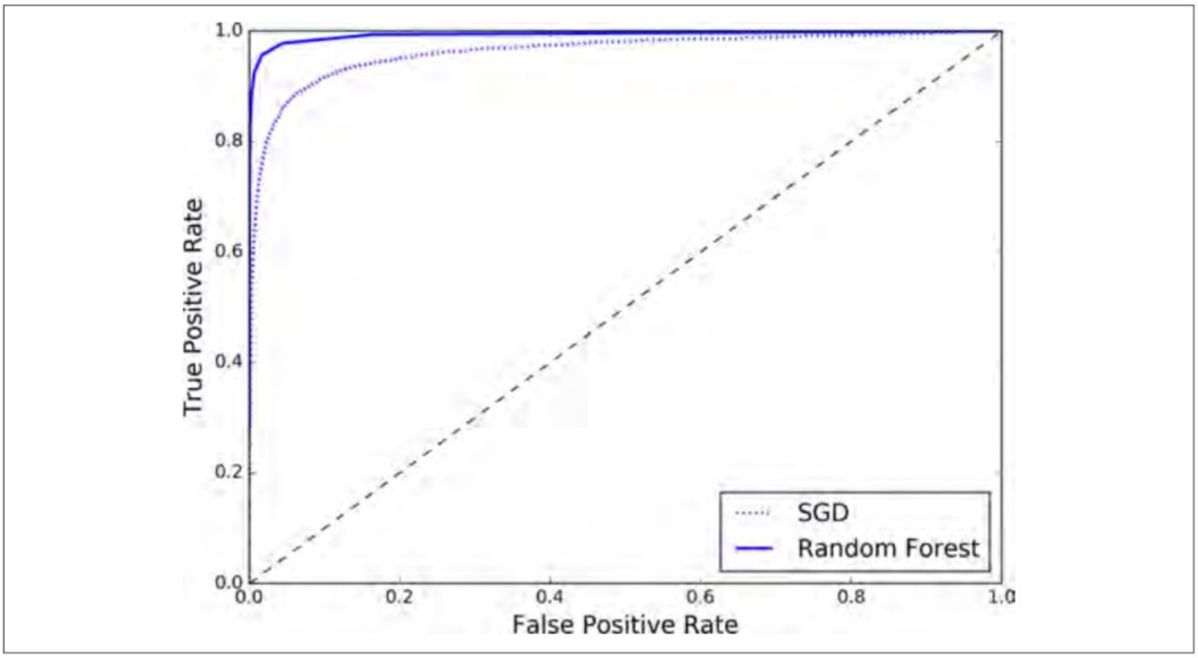


Figure 3-7. Comparing ROC curves

如你所见，`RandomForestClassifier` 的 ROC 曲线比 `SGDClassifier` 的好得多：它更靠近左上角。所以，它的 ROC AUC 也会更大。

```
>>> roc_auc_score(y_train_5, y_scores_forest)
0.99312433660038291
```

计算一下准确率和召回率：98.5% 的准确率，82.8% 的召回率。还不错。

现在你知道如何训练一个二分类器，选择合适的标准，使用交叉验证去评估你的分类器，选择满足你需要的准确率/召回率折衷方案，和比较不同模型的 ROC 曲线和 ROC AUC 数值。现在让我们检测更多的数字，而不仅仅是一个数字 5。

多类分类

二分类器只能区分两个类，而多类分类器（也被叫做多项式分类器）可以区分多于两个类。

一些算法（比如随机森林分类器或者朴素贝叶斯分类器）可以直接处理多类分类问题。其他一些算法（比如 SVM 分类器或者线性分类器）则是严格的二分类器。然后，有许多策略可以让你用二分类器去执行多类分类。

举例子，创建一个可以将图片分成 10 类（从 0 到 9）的系统的一个方法是：训练10个二分类器，每一个对应一个数字（探测器 0，探测器 1，探测器 2，以此类推）。然后当你想对某张图片进行分类的时候，让每一个分类器对这个图片进行分类，选出决策分数最高的那个分类器。这叫做“一对所有”（OvA）策略（也被叫做“一对其他”）。

另一个策略是对每一对数字都训练一个二分类器：一个分类器用来处理数字 0 和数字 1，一个用来处理数字 0 和数字 2，一个用来处理数字 1 和 2，以此类推。这叫做“一对一”（OvO）策略。如果有 N 个类。你需要训练 `N*(N-1)/2` 个分类器。对于 MNIST 问题，需要训练 45 个二分类器！当你想对一张图片进行分类，你必须将这张图片跑在全部45个二分类器上。然后看哪个类胜出。OvO 策略的主要有点是：每个分类器只需要在训练集的部分数据上面进行训练。这部分数据是它所需要区分的那两个类对应的数据。

一些算法（比如 SVM 分类器）在训练集的大小上很难扩展，所以对于这些算法，OvO 是比较好的，因为它可以在小的数据集上面可以更多地训练，较之于巨大的数据集而言。但是，对于大部分的二分类器来说，OvA 是更好的选择。

Scikit-Learn 可以探测出你想使用一个二分类器去完成多分类的任务，它会自动地执行 OvA（除了 SVM 分类器，它使用 OvO）。让我们试一下 `SGDClassifier`.

```
>>> sgd_clf.fit(X_train, y_train) # y_train, not y_train_5
>>> sgd_clf.predict([some_digit])
array([ 5.])
```

很容易。上面的代码在训练集上训练了一个 `SGDClassifier` 。这个分类器处理原始的目标 class，从 0 到 9（ `y_train` ），而不是仅仅探测是否为 5 （ `y_train_5` ）。然后它做出一个判断（在这个案例下只有一个正确的数字）。在幕后，Scikit-Learn 实际上训练了 10 个二分类器，每个分类器都产到一张图片的决策数值，选择数值最高的那个类。

为了证明这是真实的，你可以调用 `decision_function()` 方法。不是返回每个样例的一个数值，而是返回 10 个数值，一个数值对应于一个类。

```
>>> some_digit_scores = sgd_clf.decision_function([some_digit])
>>> some_digit_scores
array([[-311402.62954431, -363517.28355739, -446449.5306454 ,
        -183226.61023518, -414337.15339485, 161855.74572176,
        -452576.39616343, -471957.14962573, -518542.33997148,
        -536774.63961222]])
```

最高数值是对应于类别 5 ：

```
>>> np.argmax(some_digit_scores)
5
>>> sgd_clf.classes_
array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
>>> sgd_clf.classes_[5]
5.0
```

> 一个分类器被训练好了之后，它会保存目标类别列表到它的属性 `classes_` 中去，按照值排序。在本例子当中，在 `classes_` 数组当中的每个类的索引方便地匹配了类本身，比如，索引为 5 的类恰好是类别 5 本身。但通常不会这么幸运。

如果你想强制 Scikit-Learn 使用 OvO 策略或者 OvA 策略，你可以使用 `OneVsOneClassifier` 类或者 `OneVsRestClassifier` 类。创建一个样例，传递一个二分类器给它的构造函数。举例子，下面的代码会创建一个多类分类器，使用 OvO 策略，基于 `SGDClassifier` 。

```
>>> from sklearn.multiclass import OneVsOneClassifier
>>> ovo_clf = OneVsOneClassifier(SGDClassifier(random_state=42))
>>> ovo_clf.fit(X_train, y_train)
>>> ovo_clf.predict([some_digit])
array([ 5.])
>>> len(ovo_clf.estimators_)
45
```

训练一个 `RandomForestClassifier` 同样简单：

```
>>> forest_clf.fit(X_train, y_train)
>>> forest_clf.predict([some_digit])
array([ 5.])
```

这次 Scikit-Learn 没有必要去运行 OvO 或者 OvA，因为随机森林分类器能够直接将一个样例分到多个类别。你可以调用 `predict_proba()` ，得到样例对应的类别的概率值的列表：

```
>>> forest_clf.predict_proba([some_digit])
array([[ 0.1, 0. , 0. , 0.1, 0. , 0.8, 0. , 0. , 0. , 0. ]])
```

你可以看到这个分类器相当确信它的预测：在数组的索引 5 上的 0.8，意味着这个模型以 80% 的概率估算这张图片代表数字 5。它也认为这个图片可能是数字 0 或者数字 3，分别都是 10% 的几率。

现在当然你想评估这些分类器。像平常一样，你想使用交叉验证。让我们用 `cross_val_score()` 来评估 `SGDClassifier` 的精度。

```
>>> cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
array([ 0.84063187, 0.84899245, 0.86652998])
```

在所有测试折（test fold）上，它有 84% 的精度。如果你是用一个随机的分类器，你将会得到 10% 的正确率。所以这不是一个坏的分数，但是你可以做的更好。举例子，简单将输入正则化，将会提高精度到 90% 以上。

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler()
>>> X_train_scaled = scaler.fit_transform(X_train.astype(np.float64))
>>> cross_val_score(sgd_clf, X_train_scaled, y_train, cv=3, scoring="accuracy")
array([ 0.91011798, 0.90874544, 0.906636 ])
```

误差分析

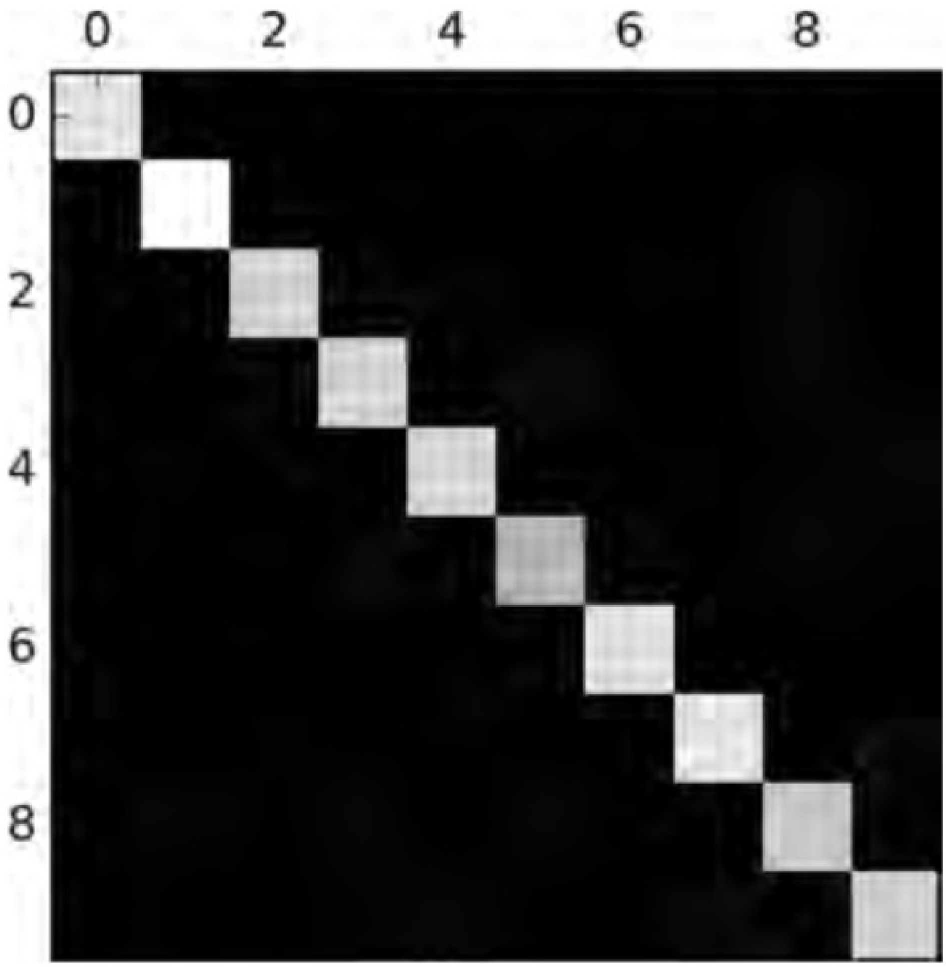
当然，如果这是一个实际的项目，你会在你的机器学习项目当中，跟随以下步骤（见附录 B）：探索准备数据的候选方案，尝试多种模型，把最好的几个模型列为入围名单，用 `GridSearchCV` 调试超参数，尽可能地自动化，像你前面的章节做的那样。在这里，我们假设你已经找到一个不错的模型，你试图找到方法去改善它。一个方式是分析模型产生的误差的类型。

首先，你可以检查混淆矩阵。你需要使用 `cross_val_predict()` 做出预测，然后调用 `confusion_matrix()` 函数，像你早前做的那样。

```
>>> y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)
>>> conf_mx = confusion_matrix(y_train, y_train_pred)
>>> conf_mx
array([[5725, 3, 24, 9, 10, 49, 50, 10, 39, 4],
        [ 2, 6493, 43, 25, 7, 40, 5, 10, 109, 8],
        [ 51, 41, 5321, 104, 89, 26, 87, 60, 166, 13],
        [ 47, 46, 141, 5342, 1, 231, 40, 50, 141, 92],
        [ 19, 29, 41, 10, 5366, 9, 56, 37, 86, 189],
        [ 73, 45, 36, 193, 64, 4582, 111, 30, 193, 94],
        [ 29, 34, 44, 2, 42, 85, 5627, 10, 45, 0],
        [ 25, 24, 74, 32, 54, 12, 6, 5787, 15, 236],
        [ 52, 161, 73, 156, 10, 163, 61, 25, 5027, 123],
        [ 43, 35, 26, 92, 178, 28, 2, 223, 82, 5240]])
```

这里是一对数字。使用 Matplotlib 的 `matshow()` 函数，将混淆矩阵以图像的方式呈现，将会更加方便。

```
plt.matshow(conf_mx, cmap=plt.cm.gray)
plt.show()
```



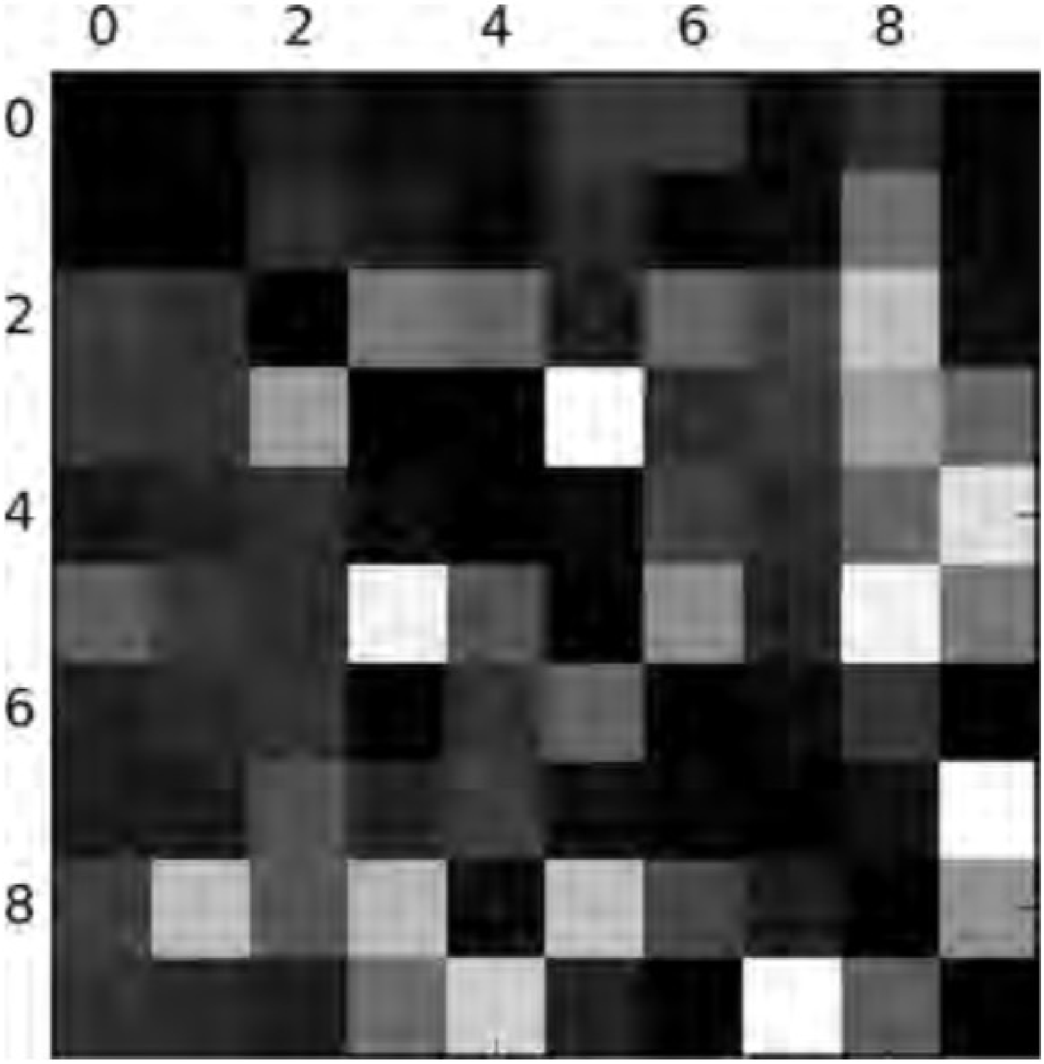
这个混淆矩阵看起来相当好，因为大多数的图片在主对角线上。在主对角线上意味着被分类正确。数字 5 对应的格子看起来比其他数字要暗淡许多。这可能是数据集当中数字 5 的图片比较少，又或者是分类器对于数字 5 的表现不如其他数字那么好。你可以验证两种情况。

让我们关注仅包含误差数据的图像呈现。首先你需要将混淆矩阵的每一个值除以相应类别的图片的总数目。这样子，你可以比较错误率，而不是绝对的错误数（这对大的类别不公平）。

```
row_sums = conf_mx.sum(axis=1, keepdims=True)
norm_conf_mx = conf_mx / row_sums
```

现在让我们用 0 来填充对角线。这样子就只保留了被错误分类的数据。让我们画出这个结果。

```
np.fill_diagonal(norm_conf_mx, 0)
plt.matshow(norm_conf_mx, cmap=plt.cm.gray)
plt.show()
```



现在你可以清楚看出分类器制造出来的各类误差。记住：行代表实际类别，列代表预测的类别。第 8、9 列相当亮，这告诉你许多图片被误分成数字 8 或者数字 9。相似的，第 8、9 行也相当亮，告诉你数字 8、数字 9 经常被误以为是其他数字。相反，一些行相当黑，比如第一行：这意味着大部分的数字 1 被正确分类（一些被误分类为数字 8 ）。留意到误差图不是严格对称的。举例子，比起将数字 8 误分类为数字 5 的数量，有更多的数字 5 被误分类为数字 8。

分析混淆矩阵通常可以给你提供深刻的见解去改善你的分类器。回顾这幅图，看样子你应该努力改善分类器在数字 8 和数字 9 上的表现，和纠正 3/5 的混淆。举例子，你可以尝试去收集更多的数据，或者你可以构造新的、有助于分类器的特征。举例子，写一个算法去数闭合的环（比如，数字 8 有两个环，数字 6 有一个， 5 没有）。又或者你可以预处理图片（比如，使用 Scikit-Learn，Pillow， OpenCV）去构造一个模式，比如闭合的环。

分析独特的误差，是获得关于你的分类器是如何工作及其为什么失败的洞见的一个好途径。但是这相对难和耗时。举例子，我们可以画出数字 3 和 5 的例子

```
cl_a, cl_b = 3, 5
X_aa = X_train[(y_train == cl_a) & (y_train_pred == cl_a)]
X_ab = X_train[(y_train == cl_a) & (y_train_pred == cl_b)]
X_ba = X_train[(y_train == cl_b) & (y_train_pred == cl_a)]
X_bb = X_train[(y_train == cl_b) & (y_train_pred == cl_b)]
plt.figure(figsize=(8,8))
plt.subplot(221); plot_digits(X_aa[:25], ../images_per_row=5)
plt.subplot(222); plot_digits(X_ab[:25], ../images_per_row=5)
plt.subplot(223); plot_digits(X_ba[:25], ../images_per_row=5)
plt.subplot(224); plot_digits(X_bb[:25], ../images_per_row=5)
plt.show()
```



左边两个 `5*5` 的块将数字识别为 3，右边的将数字识别为 5。一些被分类器错误分类的数字（比如左下角和右上角的块）是书写地相当差，甚至让人类分类都会觉得很困难（比如第 8 行第 1 列的数字 5，看起来非常像数字 3 ）。但是，大部分被误分类的数字，在我们看来都是显而易见的错误。很难明白为什么分类器会分错。原因是我们使用的简单的 `SGDClassifier` ，这是一个线性模型。它所做的全部工作就是分配一个类权重给每一个像素，然后当它看到一张新的图片，它就将加权的像素强度相加，每个类得到一个新的值。所以，因为 3 和 5 只有一小部分的像素有差异，这个模型很容易混淆它们。

3 和 5 之间的主要差异是连接顶部的线和底部的线的细线的位置。如果你画一个 3，连接处稍微向左偏移，分类器很可能将它分类成 5。反之亦然。换一个说法，这个分类器对于图片的位移和旋转相当敏感。所以，减轻 3/5 混淆的一个方法是对图片进行预处理，确保它们都很好地中心化和不过度旋转。这同样很可能帮助减轻其他类型的错误。

多标签分类

到目前为止，所有的样例都总是被分配到仅一个类。有些情况下，你也许想让你的分类器给一个样例输出多个类别。比如说，思考一个人脸识别器。如果对于同一张图片，它识别出几个人，它应该做什么？当然它应该给每一个它识别出的人贴上一个标签。比方说，这个分类器被训练成识别三个人脸，Alice，Bob，Charlie；然后当它被输入一张含有 Alice 和 Bob 的图片，它应该输出 `[1, 0, 1]`（意思是：Alice 是，Bob 不是，Charlie 是）。这种输出多个二值标签的分类系统被叫做多标签分类系统。

目前我们不打算深入脸部识别。我们可以先看一个简单点的例子，仅仅是为了阐明的目的。

```
from sklearn.neighbors import KNeighborsClassifier
y_train_large = (y_train >= 7)
y_train_odd = (y_train % 2 == 1)
y_multilabel = np.c_[y_train_large, y_train_odd]
knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train, y_multilabel)
```

这段代码创造了一个 `y_multilabel` 数组，里面包含两个目标标签。第一个标签指出这个数字是否为大数字（7，8 或者 9），第二个标签指出这个数字是否是奇数。接下来几行代码会创建一个 `KNeighborsClassifier` 样例（它支持多标签分类，但不是所有分类器都可以），然后我们使用多目标数组来训练它。现在你可以生成一个预测，然后它输出两个标签：

```
>>> knn_clf.predict([some_digit])
array([[False, True]], dtype=bool)
```

它工作正确。数字 5 不是大数（ `False` ），同时是一个奇数（ `True` ）。

有许多方法去评估一个多标签分类器，和选择正确的量度标准，这取决于你的项目。举个例子，一个方法是对每个个体标签去量度 F1 值（或者前面讨论过的其他任意的二分类器的量度标准），然后计算平均值。下面的代码计算全部标签的平均 F1 值：

```
>>> y_train_knn_pred = cross_val_predict(knn_clf, X_train, y_train, cv=3)
>>> f1_score(y_train, y_train_knn_pred, average="macro")
0.96845540180280221
```

这里假设所有标签有着同等的重要性，但可能不是这样。特别是，如果你的 Alice 的照片比 Bob 或者 Charlie 更多的时候，也许你想让分类器在 Alice 的照片上具有更大的权重。一个简单的选项是：给每一个标签的权重等于它的支持度（比如，那个标签的样例的数目）。为了做到这点，简单地在上面代码中设置 `average="weighted"` 。

多输出分类

我们即将讨论的最后一种分类任务被叫做“多输出-多类分类”（或者简称为多输出分类）。它是多标签分类的简单泛化，在这里每一个标签可以是多类别的（比如说，它可以有多于两个可能值）。

为了说明这点，我们建立一个系统，它可以去除图片当中的噪音。它将一张混有噪音的图片作为输入，期待它输出一张干净的数字图片，用一个像素强度的数组表示，就像 MNIST 图片那样。注意到这个分类器的输出是多标签的（一个像素一个标签）和每个标签可以有多个值（像素强度取值范围从 0 到 255）。所以它是一个多输出分类系统的例子。

> 分类与回归之间的界限是模糊的，比如这个例子。按理说，预测一个像素的强度更类似于一个回归任务，而不是一个分类任务。而且，多输出系统不限于分类任务。你甚至可以让你一个系统给每一个样例都输出多个标签，包括类标签和值标签。

让我们从 MNIST 的图片创建训练集和测试集开始，然后给图片的像素强度添加噪声，这里是用 NumPy 的 `randint()` 函数。目标图像是原始图像。

```
noise = rnd.randint(0, 100, (len(X_train), 784))
noise = rnd.randint(0, 100, (len(X_test), 784))
X_train_mod = X_train + noise
X_test_mod = X_test + noise
y_train_mod = X_train
y_test_mod = X_test
```

让我们看一下测试集当中的一张图片（是的，我们在窥探测试集，所以你应该马上邹眉）：



左边的加噪声的输入图片。右边是干净的目标图片。现在我们训练分类器，让它清洁这张图片：

```
knn_clf.fit(X_train_mod, y_train_mod)
clean_digit = knn_clf.predict([X_test_mod[some_index]])
plot_digit(clean_digit)
```



看起来足够接近目标图片。现在总结我们的分类之旅。希望你现在应该知道如何选择好的量度标准，挑选出合适的准确率/召回率的折衷方案，比较分类器，更概括地说，就是为不同的任务建立起好的分类系统。

练习

1. 尝试在 MNIST 数据集上建立一个分类器，使它在测试集上的精度超过 97%。提示： `KNeighborsClassifier` 非常适合这个任务。你只需要找出一个好的超参数值（试一下对权重和超参数 `n_neighbors` 进行网格搜索）。
2. 写一个函数可以是 MNIST 中的图像任意方向移动（上下左右）一个像素。然后，对训练集上的每张图片，复制四个移动后的副本（每个方向一个副本），把它们加到训练集当中去。最后在扩展后的训练集上训练你最好的模型，并且在测试集上测量它的精度。你应该会观察到你的模型会有更好的表现。这种人工扩大训练集的方法叫做数据增强，或者训练集扩张。
3. 拿 Titanic 数据集去捣鼓一番。开始这个项目有一个很棒的平台：Kaggle！
4. 建立一个垃圾邮件分类器（这是一个更有挑战性的练习）：
    - 下载垃圾邮件和非垃圾邮件的样例数据。地址是Apache SpamAssassin 的公共数据集
    - 解压这些数据集，并且熟悉它的数据格式。
    - 将数据集分成训练集和测试集
    - 写一个数据准备的流水线，将每一封邮件转换为特征向量。你的流水线应该将一封邮件转换为一个稀疏向量，对于所有可能的词，这个向量标志哪个词出现了，哪个词没有出现。举例子，如果所有邮件只包含了 `"Hello","How","are", "you"` 这四个词，那么一封邮件（内容是： `"Hello you Hello Hello you"` ）将会被转换为向量 `[1, 0, 0, 1]` (意思是： `"Hello"` 出现， `"How"` 不出现， `"are"` 不出现， `"you"` 出现)，或者 `[3, 0, 0, 2]` ，如果你想数出每个单词出现的次数。

- 你也许想给你的流水线增加超参数，控制是否剥过邮件头、将邮件转换为小写、去除标点符号、将所有 URL 替换成`"URL"`，将所有数字替换成`"NUMBER"`，或者甚至提取词干（比如，截断词尾。有现成的 Python 库可以做到这点）。
- 然后 尝试几个不同的分类器，看看你可否建立一个很棒的垃圾邮件分类器，同时有着高召回率和高准确率。

第4章 训练模型

在之前的描述中，我们通常把机器学习模型和训练算法当作黑箱子来处理。如果你实践过前几章的一些示例，你惊奇的发现你可以优化回归系统，改进数字图像的分类器，你甚至可以零基础搭建一个垃圾邮件的分类器，但是你却对它们内部的工作流程一无所知。事实上，许多场合你都不需要知道这些黑箱子的内部有什么，干了什么。

然而，如果你对其内部的工作流程有一定了解的话，当面对一个机器学习任务时候，这些理论可以帮助你快速的找到恰当的机器学习模型，合适的训练算法，以及一个好的假设集。同时，了解黑箱子内部的构成，有助于你更好地调试参数以及更有效的误差分析。本章讨论的大部分话题对于机器学习模型的理解，构建，以及神经网络（详细参考本书的第二部分）的训练都是非常重要的。

首先我们将以一个简单的线性回归模型为例，讨论两种不同的训练方法来得到模型的最优解：

- 直接使用封闭方程进行求根运算，得到模型在当前训练集上的最优参数（即在训练集上使损失函数达到最小值的模型参数）
- 使用迭代优化方法：梯度下降（GD），在训练集上，它可以逐渐调整模型参数以获得最小的损失函数，最终，参数会收敛到和第一种方法相同的的值。同时，我们也会介绍一些梯度下降的变体形式：批量梯度下降（Batch GD）、小批量梯度下降（Mini-batch GD）、随机梯度下降（Stochastic GD），在第二部分的神经网络部分，我们会多次使用它们。

接下来，我们将研究一个更复杂的模型：多项式回归，它可以拟合非线性数据集，由于它比线性模型拥有更多的参数，于是它更容易出现模型的过拟合。因此，我们将介绍如何通过学习曲线去判断模型是否出现了过拟合，并介绍几种正则化方法以减少模型出现过拟合的风险。

最后，我们将介绍两个常用于分类的模型：Logistic回归和Softmax回归

> 提示
>
> 在本章中包含许多数学公式，以及一些线性代数和微积分基本概念。为了理解这些公式，你需要知道什么是向量，什么是矩阵，以及它们直接是如何转化的，以及什么是点积，什么是矩阵的逆，什么是偏导数。如果你对这些不是很熟悉的话，你可以阅读本书提供的 Jupyter 在线笔记，它包括了线性代数和微积分的入门指导。对于那些不喜欢数学的人，你也应该快速简单的浏览这些公式。希望它足以帮助你理解大多数的概念。

线性回归

在第一章，我们介绍了一个简单的生活满意度回归模型:

$$life_satisfaction = \theta_0 + \theta_1 * GDP_per_capita$$

这个模型仅仅是输入量 `GDP_per_capita` 的线性函数，θ_0 和 θ_1 是这个模型的参数，线性模型更一般化的描述指通过计算输入变量的加权和，并加上一个常数偏置项（截距项）来得到一个预测值。如公式 4-1：

公式 4-1：线性回归预测模型

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

- $\hat{y}$ 表示预测结果
- n 表示特征的个数
- x_i 表示第 `i` 个特征的值
- θ_j 表示第 `j` 个参数（包括偏置项 θ_0 和特征权重值 $\theta_1, \theta_2, \ldots, \theta_n$）

上述公式可以写成更为简洁的向量形式，如公式 4-2：

公式 4-2：线性回归预测模型（向量形式）

$$\hat{y} = h_\theta(\mathbf{x}) = \theta^T \cdot \mathbf{x}$$

- θ 表示模型的参数向量包括偏置项 θ_0 和特征权重值 θ_1 到 θ_n
- θ^T 表示向量θ的转置（行向量变为了列向量）
- $\mathbf{x}$ 为每个样本中特征值的向量形式，包括 x_1 到 x_n，而且 x_0 恒为 1
- $\theta^T \cdot \mathbf{x}$ 表示 θ^T 和$\mathbf{x}$ 的点积
- h_θ 表示参数为 θ 的假设函数

怎么样去训练一个线性回归模型呢？好吧，回想一下，训练一个模型指的是设置模型的参数使得这个模型在训练集的表现较好。为此，我们首先需要找到一个衡量模型好坏的评定方法。在第二章，我们介绍到在回归模型上，最常见的评定标准是均方根误差（RMSE，详见公式 2-1）。因此，为了训练一个线性回归模型，你需要找到一个 θ 值，它使得均方根误差（标准误差）达到最小值。实践过程中，最小化均方误差比最小化均方根误差更加的简单，这两个过程会得到相同的 θ，因为函数在最小值时候的自变量，同样能使函数的方根运算得到最小值。

在训练集 $\mathbf{X}$ 上使用公式 4-3 来计算线性回归假设 h_θ 的均方差（MSE）。

公式 4-3：线性回归模型的 MSE 损失函数

$$MSE(\mathbf{X}, h_\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)}\right)^2$$

公式中符号的含义大多数都在第二章（详见“符号”）进行了说明，不同的是：为了突出模型的参数向量 θ，使用 h_θ 来代替 h。以后的使用中为了公式的简洁，使用 $MSE(\theta)$ 来代替 $MSE(\mathbf{X}, h_\theta)$。

正态方程

为了找到最小化损失函数的 θ 值，可以采用公式解，换句话说，就是可以通过解正态方程直接得到最后的结果。

公式 4-4：正态方程

$$\hat{\theta} = (\mathbf{X}^T \cdot \mathbf{X})^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

- $\hat{\theta}$ 指最小化损失 θ 的值
- $\mathbf{y}$ 是一个向量，其包含了 $y^{(1)}$ 到 $y^{(m)}$ 的值

让我们生成一些近似线性的数据（如图 4-1）来测试一下这个方程。

```
import numpy as np
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```

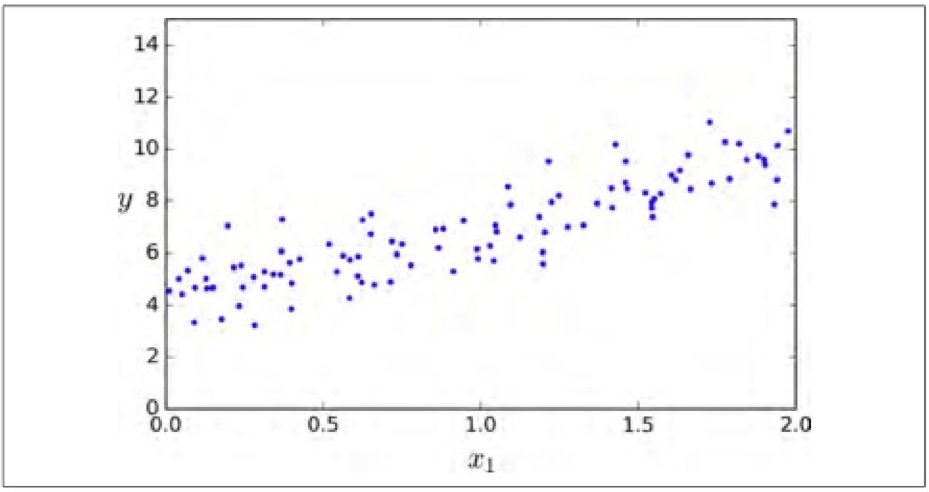


图 4-1：随机线性数据集

现在让我们使用正态方程来计算 $\hat{\theta}$，我们将使用 Numpy 的线性代数模块（`np.linalg`）中的 `inv()` 函数来计算矩阵的逆，以及 `dot()` 方法来计算矩阵的乘法。

```
X_b = np.c_[np.ones((100, 1)), X]
theta_best = np.linalg.inv(X_b.T.dot(X_B)).dot(X_b.T).dot(y)
```

我们生产数据的函数实际上是 $y = 4 + 3x_0 +$。让我们看一下最后的计算结果。

```
>>> theta_best
array([[4.21509616],[2.77011339]])
```

我们希望最后得到的参数为 $\theta_0 = 4, \theta_1 = 3$ 而不是 $\theta_0 = 3.865, \theta_1 = 3.139$（译者注：我认为应该是 $\theta_0 = 4.2150, \theta_1 = 2.7701$）。这已经足够了，由于存在噪声，参数不可能达到到原始函数的值。

现在我们能够使用 $\hat{\theta}$ 来进行预测：

```
>>> X_new = np.array([[0],[2]])
>>> X_new_b = np.c_[np.ones((2, 1)), X_new]
>>> y_predict = X_new_b.dot(theta.best)
>>> y_predict
array([[4.21509616],[9.75532293]])
```

画出这个模型的图像，如图 4-2

```
plt.plot(X_new,y_predict,"r-")
plt.plot(X,y,"b.")
plt.axis([0,2,0,15])
plt.show()
```

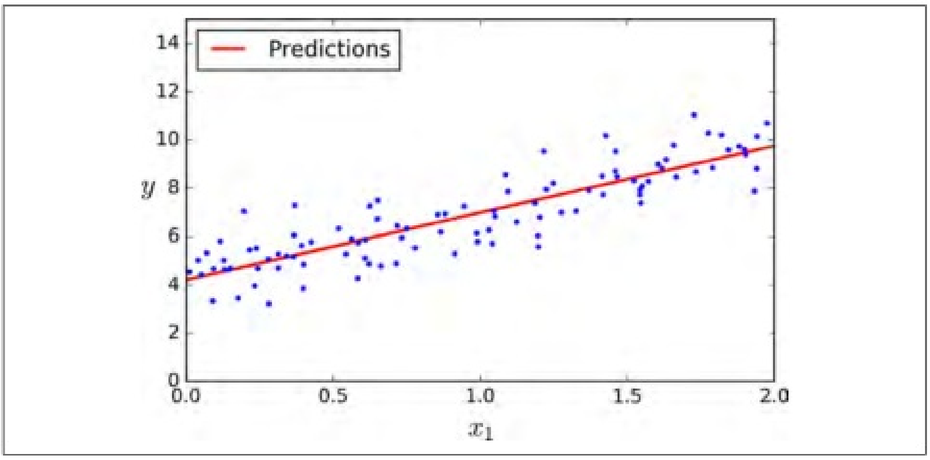


图4-2：线性回归预测

使用下面的 Scikit-Learn 代码可以达到相同的效果：

```
>>> form sklearn.linear_model import LinearRegression
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X,y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([4.21509616]),array([2.77011339]))
>>> lin_reg.predict(X_new)
array([[4.21509616],[9.75532293]])
```

计算复杂度

正态方程需要计算矩阵 $\mathbf{X}^T \cdot \mathbf{X}$ 的逆，它是一个 $n * n$ 的矩阵（n 是特征的个数）。这样一个矩阵求逆的运算复杂度大约在 $O(n^{2.4})$ 到 $O(n^3)$ 之间，具体值取决于计算方式。换句话说，如果你将你的特征个数翻倍的话，其计算时间大概会变为原来的 5.3（$2^{2.4}$）到 8（2^3）倍。

> 提示
>
> 当特征的个数较大的时候（例如：特征数量为 100000），正态方程求解将会非常慢。

有利的一面是，这个方程在训练集上对于每一个实例来说是线性的，其复杂度为 $O(m)$，因此只要有能放得下它的内存空间，它就可以对大规模数据进行训练。同时，一旦你得到了线性回归模型（通过解正态方程或者其他的算法），进行预测是非常快的。因为模型中计算复杂度对于要进行预测的实例数量和特征个数都是线性的。 换句话说，当实例个数变为原来的两倍多的时候（或特征个数变为原来的两倍多），预测时间也仅仅是原来的两倍多。

接下来，我们将介绍另一种方法去训练模型。这种方法适合在特征个数非常多，训练实例非常多，内存无法满足要求的时候使用。

梯度下降

梯度下降是一种非常通用的优化算法，它能够很好地解决一系列问题。梯度下降的整体思路是通过的迭代来逐渐调整参数使得损失函数达到最小值。

假设浓雾下，你迷失在了大山中，你只能感受到自己脚下的坡度。为了最快到达山底，一个最好的方法就是沿着坡度最陡的地方下山。这其实就是梯度下降所做的：它计算误差函数关于参数向量θ的局部梯度，同时它沿着梯度下降的方向进行下一次迭代。当梯度值为零的时候，就达到了误差函数最小值 。

具体来说，开始时，需要选定一个随机的θ（这个值称为随机初始值），然后逐渐去改进它，每一次变化一小步，每一步都试着降低损失函数（例如：均方差损失函数），直到算法收敛到一个最小值（如图：4-3）。

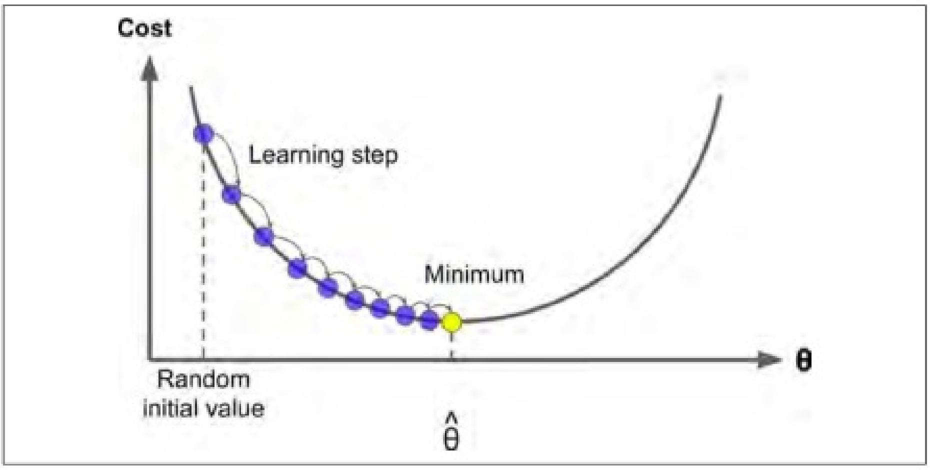


图 4-3：梯度下降

在梯度下降中一个重要的参数是步长，超参数学习率的值决定了步长的大小。如果学习率太小，必须经过多次迭代，算法才能收敛，这是非常耗时的（如图 4-4）。

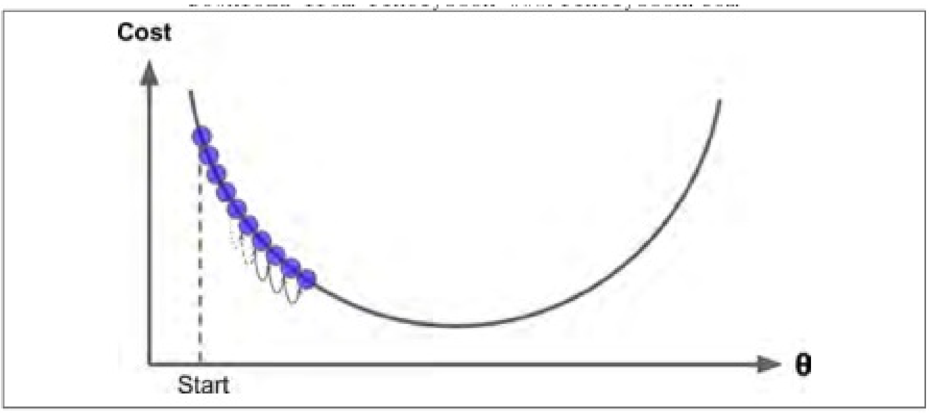


图 4-4:学习率过小

另一方面，如果学习率太大，你将跳过最低点，到达山谷的另一面，可能下一次的值比上一次还要大。这可能使的算法是发散的，函数值变得越来越大，永远不可能找到一个好的答案（如图 4-5）。

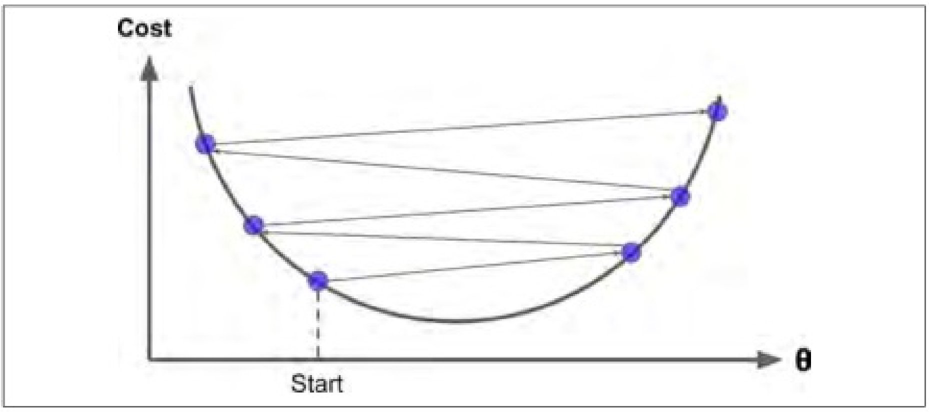


图 4-5：学习率过大

最后，并不是所有的损失函数看起来都像一个规则的碗。它们可能是洞，山脊，高原和各种不规则的地形，使它们收敛到最小值非常的困难。 图 4-6 显示了梯度下降的两个主要挑战：如果随机初始值选在了图像的左侧，则它将收敛到局部最小值，这个值要比全局最小值要大。 如果它从右侧开始，那么跨越高原将需要很长时间，如果你早早地结束训练，你将永远到不了全局最小值。

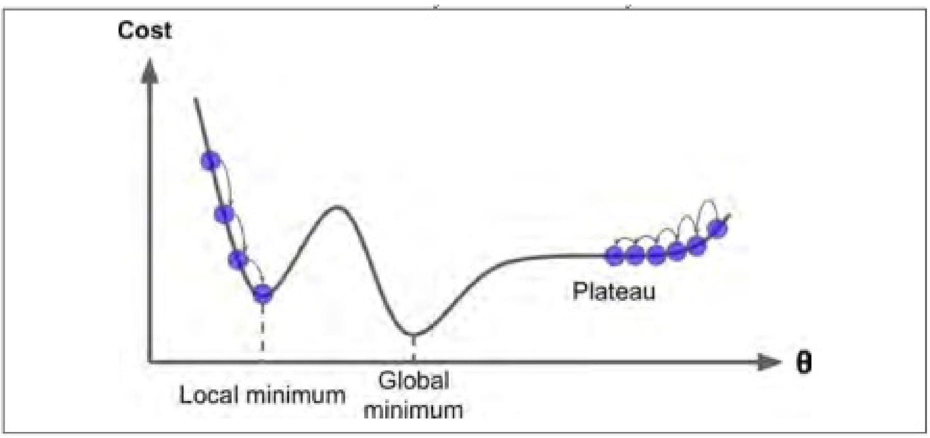


图 4-6：梯度下降的陷阱

幸运的是线性回归模型的均方差损失函数是一个凸函数，这意味着如果你选择曲线上的任意两点，它们的连线段不会与曲线发生交叉（译者注：该线段不会与曲线有第三个交点）。这意味着这个损失函数没有局部最小值，仅仅只有一个全局最小值。同时它也是一个斜率不能突变的连续函数。这两个因素导致了一个好的结果：梯度下降可以无限接近全局最小值。（只要你训练时间足够长，同时学习率不是太大 ）。

事实上，损失函数的图像呈现碗状，但是不同特征的取值范围相差较大的时，这个碗可能是细长的。图 4-7 展示了梯度下降在不同训练集上的表现。在左图中，特征 1 和特征 2 有着相同的数值尺度。在右图中，特征 1 比特征2的取值要小的多，由于特征 1 较小，因此损失函数改变时，θ_1 会有较大的变化，于是这个图像会在θ_1轴方向变得细长。

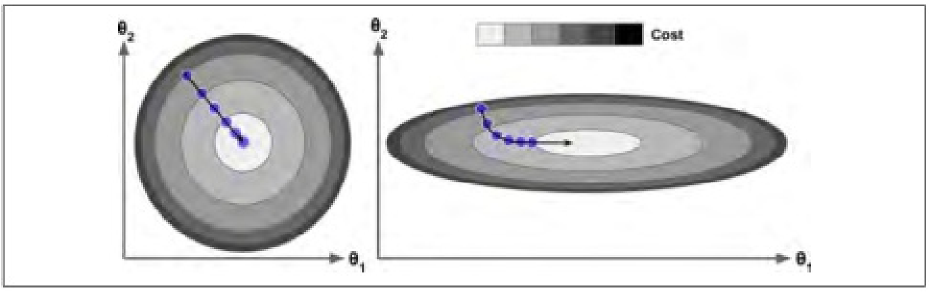


图 4-7：有无特征缩放的梯度下降

正如你看到的，左面的梯度下降可以直接快速地到达最小值，然而在右面的梯度下降第一次前进的方向几乎和全局最小值的方向垂直，并且最后到达一个几乎平坦的山谷，在平坦的山谷走了很长时间。它最终会达到最小值，但它需要很长时间。

> 提示
>
> 当我们使用梯度下降的时候，应该确保所有的特征有着相近的尺度范围（例如：使用 Scikit Learn 的 `StandardScaler`类），否则它将需要很长的时间才能够收敛。

这幅图也表明了一个事实：训练模型意味着找到一组模型参数，这组参数可以在训练集上使得损失函数最小。这是对于模型参数空间的搜索，模型的参数越多，参数空间的维度越多，找到合适的参数越困难。例如在300维的空间找到一枚针要比在三维空间里找到一枚针复杂的多。幸运的是线性回归模型的损失函数是凸函数，这个最优参数一定在碗的底部。

批量梯度下降

使用梯度下降的过程中，你需要计算每一个 θ_j 下损失函数的梯度。换句话说，你需要计算当θ_j变化一点点时，损失函数改变了多少。这称为偏导数，它就像当你面对东方的时候问："我脚下的坡度是多少？"。然后面向北方的时候问同样的问题（如果你能想象一个超过三维的宇宙，可以对所有的方向都这样做）。公式 4-5 计算关于 θ_j 的损失函数的偏导数，记为 $\frac{\partial}{\partial\theta_j}MSE(\theta)$ 。

公式 4-5： 损失函数的偏导数

$$\frac{\partial}{\partial \theta_j} MSE(\theta) = \frac{2}{m}\sum_{i=1}^{m}(\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)}){x_j}^{(i)}$$

为了避免单独计算每一个梯度，你也可以使用公式 4-6 来一起计算它们。梯度向量记为 $\nabla_\theta MSE(\theta)$，其包含了损失函数所有的偏导数（每个模型参数只出现一次）。

公式 4-6：损失函数的梯度向量

$$\nabla_\theta MSE(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} MSE(\theta) \\ \frac{\partial}{\partial \theta_1} MSE(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} MSE(\theta) \end{pmatrix} = \frac{2}{m}\mathbf{X}^T \cdot (\mathbf{X} \cdot \theta - y)$$

> 提示
>
> 在这个方程中每一步计算时都包含了整个训练集 $\mathbf{X}$，这也是为什么这个算法称为批量梯度下降：每一次训练过程都使用所有的的训练数据。因此，在大数据集上，其会变得相当的慢（但是我们接下来将会介绍更快的梯度下降算法）。然而，梯度下降的运算规模和特征的数量成正比。训练一个数千数量特征的线性回归模型使用*梯度下降要比使用正态方程快的多。

一旦求得了方向是上山的梯度向量，你就可以向着相反的方向去下山。这意味着从 θ 中减去 $\nabla_\theta MSE(\theta)$。学习率 η 和梯度向量的积决定了下山时每一步的大小，如公式 4-7。

公式 4-7：梯度下降步长

$$\theta^{(next\ step)} = \theta - \eta\nabla_\theta MSE(\theta)$$

让我们看一下这个算法的应用：

```
eta = 0.1 # 学习率
n_iterations = 1000
m = 100

theta = np.random.randn(2,1) # 随机初始值

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradiens
```

这不是太难，让我们看一下最后的结果 θ：

```
>>> theta
array([[4.21509616],[2.77011339]])
```

看！正态方程的表现非常好。完美地求出了梯度下降的参数。但是当你换一个学习率会发生什么？图 4-8 展示了使用了三个不同的学习率进行梯度下降的前 10 步运算（虚线代表起始位置）。

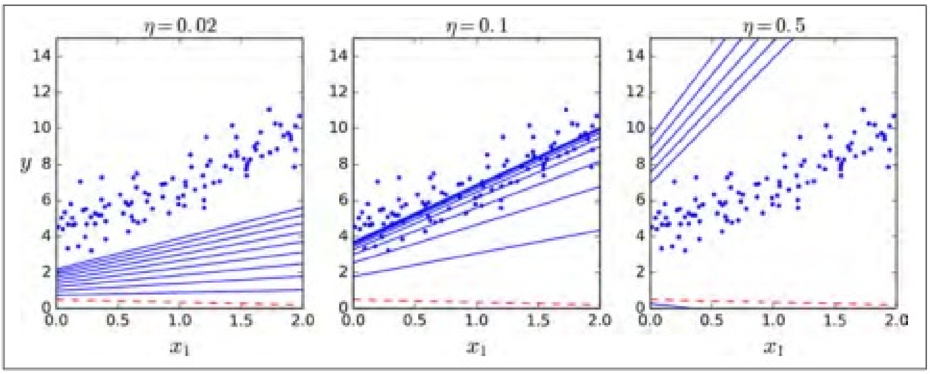


图 4-8：不同学习率的梯度下降

在左面的那副图中，学习率是最小的，算法几乎不能求出最后的结果，而且还会花费大量的时间。在中间的这幅图中，学习率的表现看起来不错，仅仅几次迭代后，它就收敛到了最后的结果。在右面的那副图中，学习率太大了，算法是发散的，跳过了所有的训练样本，同时每一步都离正确的结果越来越远。

为了找到一个好的学习率，你可以使用网格搜索（详见第二章）。当然，你一般会限制迭代的次数，以便网格搜索可以消除模型需要很长时间才能收敛这一个问题。

你可能想知道如何选取迭代的次数。如果它太小了，当算法停止的时候，你依然没有找到最优解。如果它太大了，算法会非常的耗时同时后来的迭代参数也不会发生改变。一个简单的解决方法是：设置一个非常大的迭代次数，但是当梯度向量变得非常小的时候，结束迭代。非常小指的是：梯度向量小于一个值 ε（称为容差）。这时候可以认为梯度下降几乎已经达到了最小值。

> 收敛速率：
>
> 当损失函数是凸函数，同时它的斜率不能突变（就像均方差损失函数那样），那么它的批量梯度下降算法固定学习率之后，它的收敛速率是 $O(\frac{1}{iterations})$。换句话说，如果你将容差 ε 缩小 10 倍后（这样可以得到一个更精确的结果），这个算法的迭代次数大约会变成原来的 10 倍。

随机梯度下降

批量梯度下降的最要问题是计算每一步的梯度时都需要使用整个训练集，这导致在规模较大的数据集上，其会变得非常的慢。与其完全相反的随机梯度下降，在每一步的梯度计算上只随机选取训练集中的一个样本。很明显，由于每一次的操作都使用了非常少的数据，这样使得算法变得非常快。由于每一次迭代，只需要在内存中有一个实例，这使随机梯度算法可以在大规模训练集上使用。

另一方面，由于它的随机性，与批量梯度下降相比，其呈现出更多的不规律性：它到达最小值不是平缓的下降，损失函数会忽高忽低，只是在大体上呈下降趋势。随着时间的推移，它会非常的靠近最小值，但是它不会停止在一个值上，它会一直在这个值附近摆动（如图 4-9）。因此，当算法停止的时候，最后的参数还不错，但不是最优值。

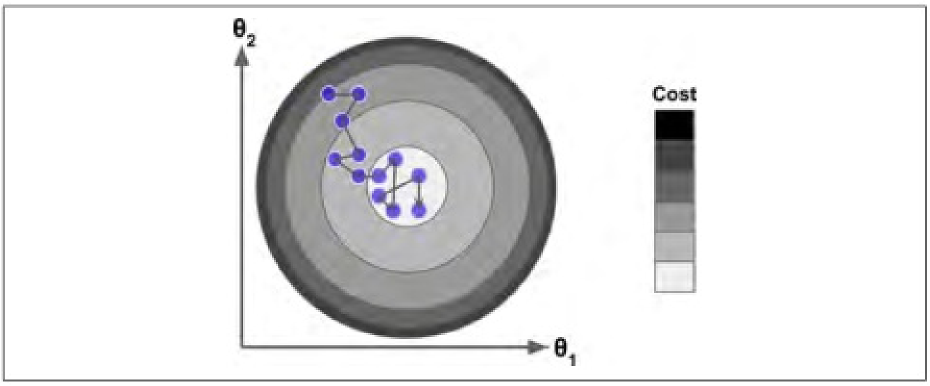


图4-9：随机梯度下降

当损失函数很不规则时（如图 4-6），随机梯度下降算法能够跳过局部最小值。因此，随机梯度下降在寻找全局最小值上比批量梯度下降表现要好。

虽然随机性可以很好的跳过局部最优值，但同时它却不能达到最小值。解决这个难题的一个办法是逐渐降低学习率。 开始时，走的每一步较大（这有助于快速前进同时跳过局部最小值），然后变得越来越小，从而使算法到达全局最小值。 这个过程被称为模拟退火，因为它类似于熔融金属慢慢冷却的冶金学退火过程。 决定每次迭代的学习率的函数称为 `learning schedule` 。 如果学习速度降低得过快，你可能会陷入局部最小值，甚至在到达最小值的半路就停止了。 如果学习速度降低得太慢，你可能在最小值的附近长时间摆动，同时如果过早停止训练，最终只会出现次优解。

下面的代码使用一个简单的 `learning schedule` 来实现随机梯度下降：

```
n_epochs = 50
t0, t1 = 5, 50  #learning_schedule的超参数

def learning_schedule(t):
    return t0 / (t + t1)

theta = np.random.randn(2,1)

for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi,dot(theta)-yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradiens
```

按习惯来讲，我们进行 m 轮的迭代，每一轮迭代被称为一代。在整个训练集上，随机梯度下降迭代了 1000 次时，一般在第 50 次的时候就可以达到一个比较好的结果。

```
>>> theta
array([[4.21076011],[2.748560791]])
```

图 4-10 展示了前 10 次的训练过程（注意每一步的不规则程度）。

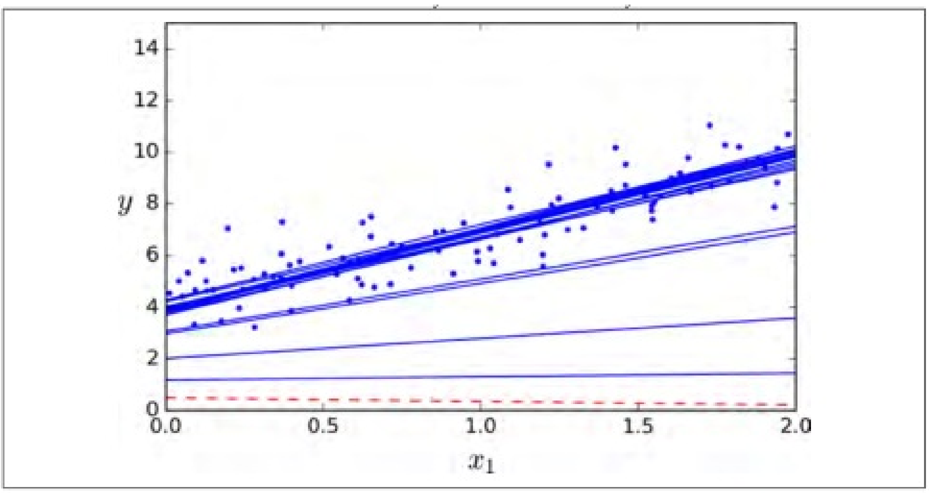


图 4-10：随机梯度下降的前10次迭代

由于每个实例的选择是随机的，有的实例可能在每一代中都被选到，这样其他的实例也可能一直不被选到。如果你想保证每一代迭代过程，算法可以遍历所有实例，一种方法是将训练集打乱重排，然后选择一个实例，之后再继续打乱重排，以此类推一直进行下去。但是这样收敛速度会非常的慢。

通过使用 Scikit-Learn 完成线性回归的随机梯度下降，你需要使用 `SGDRegressor` 类，这个类默认优化的是均方差损失函数。下面的代码迭代了 50 代，其学习率 η 为0.1（`eta0=0.1`），使用默认的 `learning schedule`（与前面的不一样），同时也没有添加任何正则项（`penalty = None`）：

```
from sklearn.linear_model import SGDRegressor
sgd_reg + SGDRregressor(n_iter=50, penalty=None, eta0=0.1)
sgd_reg.fit(X,y.ravel())
```

你可以再一次发现，这个结果非常的接近正态方程的解：

```
>>> sgd_reg.intercept_, sgd_reg.coef_
(array([4.18380366]),array([2.74205299]))
```

小批量梯度下降

最后一个梯度下降算法，我们将介绍小批量梯度下降算法。一旦你理解了批量梯度下降和随机梯度下降，再去理解小批量梯度下降是非常简单的。在迭代的每一步，批量梯度使用整个训练集，随机梯度时候用仅仅一个实例，在小批量梯度下降中，它则使用一个随机的小型实

例集。它比随机梯度的主要优点在于你可以通过矩阵运算的硬件优化得到一个较好的训练表现，尤其当你使用 GPU 进行运算的时候。

小批量梯度下降在参数空间上的表现比随机梯度下降要好的多，尤其在有大量的小型实例集时。作为结果，小批量梯度下降会比随机梯度更靠近最小值。但是，另一方面，它有可能陷在局部最小值中（在遇到局部最小值问题的情况下，和我们之前看到的线性回归不一样）。图4-11显示了训练期间三种梯度下降算法在参数空间中所采用的路径。 他们都接近最小值，但批量梯度的路径最后停在了最小值，而随机梯度和小批量梯度最后都在最小值附近摆动。但是，不要忘记，批次梯度需要花费大量时间来完成每一步，但是，如果你使用了一个较好的 `learning schedule` ，随机梯度和小批量梯度也可以得到最小值。

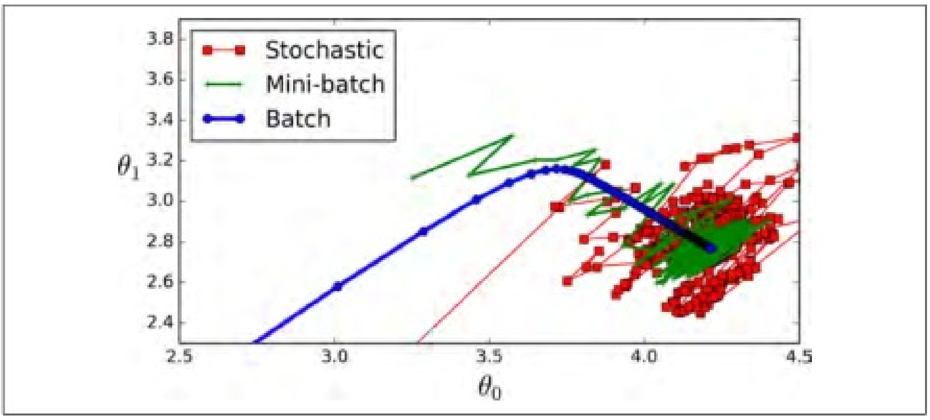


图 4-11：参数空间的梯度下降路径

让我比较一下目前我们已经探讨过的对线性回归的梯度下降算法。如表 4-1 所示，其中 m 表示训练样本的个数，n 表示特征的个数。

表 4-1：比较线性回归的不同梯度下降算法

Algorithm	Large *m*	Out-of-core support	Large *n*	Hyperparams	Scaling required	Scikit-Learn
Normal Equation	Fast	No	Slow	0	No	`LinearRegression`
Batch GD	Slow	No	Fast	2	Yes	n/a
Stochastic GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`
Mini-batch GD	Fast	Yes	Fast	≥2	Yes	n/a

> 提示
>
> 上述算法在完成训练后，得到的参数基本没什么不同，它们会得到非常相似的模型，最后会以一样的方式去进行预测。

多项式回归

如果你的数据实际上比简单的直线更复杂呢？ 令人惊讶的是，你依然可以使用线性模型来拟合非线性数据。 一个简单的方法是对每个特征进行加权后作为新的特征，然后训练一个线性模型在这个扩展的特征集。 这种方法称为多项式回归。

让我们看一个例子。 首先，我们根据一个简单的二次方程（并加上一些噪声，如图 4-12）来生成一些非线性数据：

```
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```

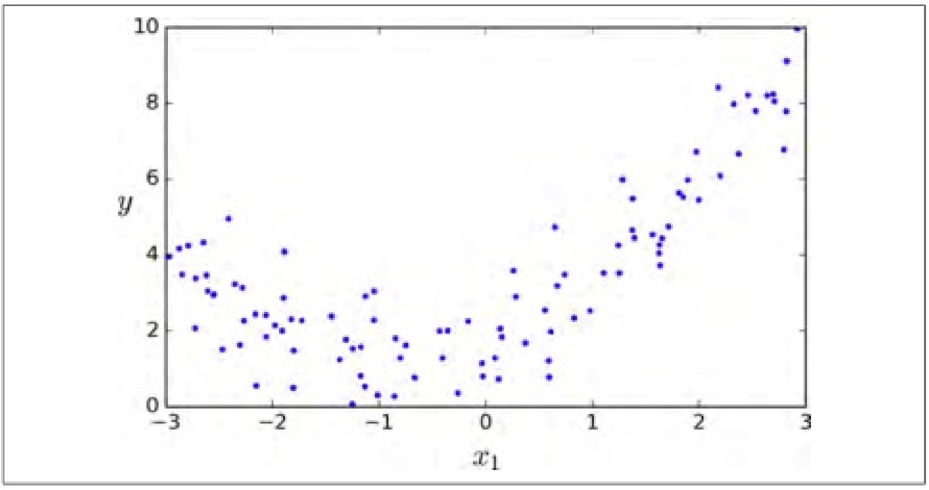


图 4-12：生产加入噪声的非线性数据

很清楚的看出，直线不能恰当的拟合这些数据。于是，我们使用 Scikit-Learning 的 `PolynomialFeatures` 类进行训练数据集的转换，让训练集中每个特征的平方（2 次多项式）作为新特征（在这种情况下，仅存在一个特征）：

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features = PolynomialFeatures(degree=2,include_bias=False)
>>> X_poly = poly_features.fit_transform(X)
>>> X[0]
array([-0.75275929])
>>> X_poly[0]
array([-0.75275929, 0.56664654])
```

`X_poly` 现在包含原始特征X并加上了这个特征的平方 X^2。现在你可以在这个扩展训练集上使用 `LinearRegression` 模型进行拟合，如图 4-13：

```
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([ 1.78134581]), array([[ 0.93366893, 0.56456263]]))
```

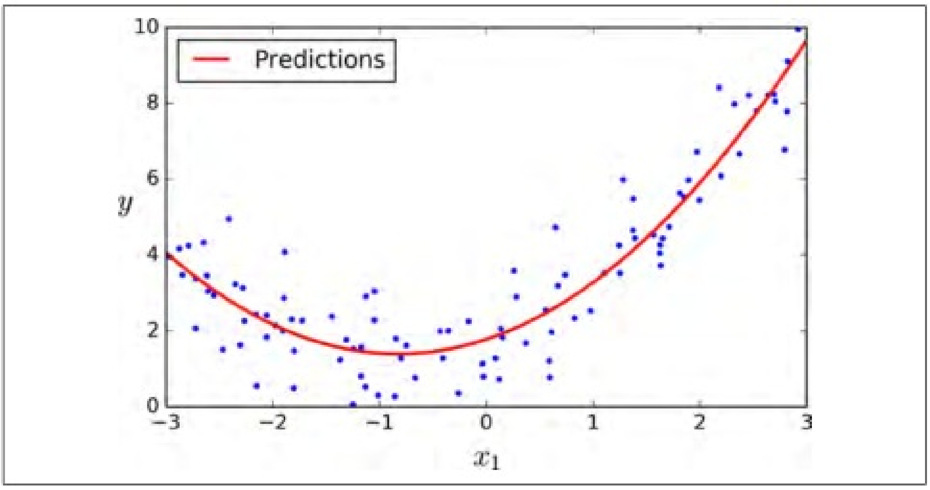


图 4-13：多项式回归模型预测

还是不错的，模型预测函数 $\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$，事实上原始函数为 $y = 0.5x_1^2 + 1.0x_1 + 2.0$ 再加上一些高斯噪声。

请注意，当存在多个特征时，多项式回归能够找出特征之间的关系（这是普通线性回归模型无法做到的）。这是因为 `LinearRegression` 会自动添加当前阶数下特征的所有组合。例如，如果有两个特征 a, b，使用 3 阶（`degree=3`）的 `LinearRegression` 时，不仅有 a^2, a^3, b^2 以及 b^3，同时也会有它们的其他组合项 ab, a^2b, ab^2。

> 提示
>
> `PolynomialFeatures(degree=d)` 把一个包含 n 个特征的数组转换为一个包含 $\frac{(n+d)!}{d!n!}$ 特征的数组，$n!$ 表示 n 的阶乘，等于 $1 * 2 * 3 \cdots * n$。小心大量特征的组合爆炸！

学习曲线

如果你使用一个高阶的多项式回归，你可能发现它的拟合程度要比普通的线性回归要好的多。例如，图 4-14 使用一个 300 阶的多项式模型去拟合之前的数据集，并同简单线性回归、2 阶的多项式回归进行比较。注意 300 阶的多项式模型如何摆动以尽可能接近训练实例。

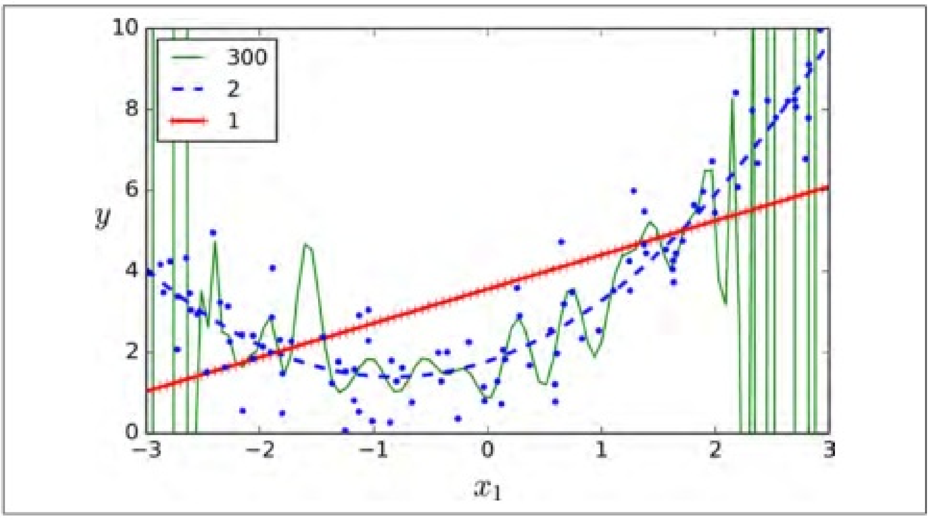


图 4-14：高阶多项式回归

当然，这种高阶多项式回归模型在这个训练集上严重过拟合了，线性模型则欠拟合。在这个训练集上，二次模型有着较好的泛化能力。那是因为在生成数据时使用了二次模型，但是一般我们不知道这个数据生成函数是什么，那我们该如何决定我们模型的复杂度呢？你如何告诉我你的模型是过拟合还是欠拟合？

在第二章，你可以使用交叉验证来估计一个模型的泛化能力。如果一个模型在训练集上表现良好，通过交叉验证指标却得出其泛化能力很差，那么你的模型就是过拟合了。如果在这两方面都表现不好，那么它就是欠拟合了。这种方法可以告诉我们，你的模型是太复杂还是太简单了。

另一种方法是观察学习曲线：画出模型在训练集上的表现，同时画出以训练集规模为自变量的训练集函数。为了得到图像，需要在训练集的不同规模子集上进行多次训练。下面的代码定义了一个函数，用来画出给定训练集后的模型学习曲线：

```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

def plot_learning_curves(model, X, y):
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)
    train_errors, val_errors = [], []
    for m in range(1, len(X_train)):
        model.fit(X_train[:m], y_train[:m])
        y_train_predict = model.predict(X_train[:m])
        y_val_predict = model.predict(X_val)
        train_errors.append(mean_squared_error(y_train_predict, y_train[:m]))
        val_errors.append(mean_squared_error(y_val_predict, y_val))
plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")
```

我们一起看一下简单线性回归模型的学习曲线（图 4-15）：

```
lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)
```

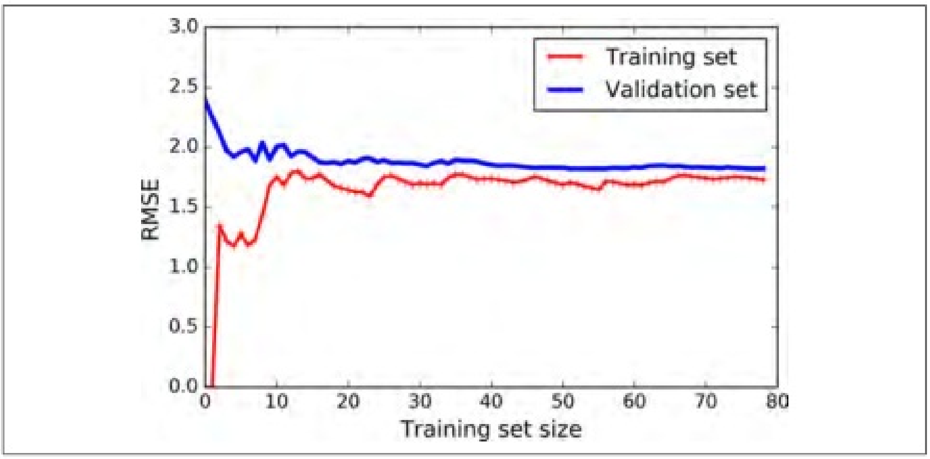


图 4-15：学习曲线

这幅图值得我们深究。首先，我们观察训练集的表现：当训练集只有一两个样本的时候，模型能够非常好的拟合它们，这也是为什么曲线是从零开始的原因。但是当加入了一些新的样本的时候，训练集上的拟合程度变得难以接受，出现这种情况有两个原因，一是因为数据中含有噪声，另一个是数据根本不是线性的。因此随着数据规模的增大，误差也会一直增大，直到达到高原地带并趋于稳定，在之后，继续加入新的样本，模型的平均误差不会变得更好或者更差。我们继续来看模型在验证集上的表现，当以非常少的样本去训练时，模型不能恰当的泛化，也就是为什么验证误差一开始是非常大的。当训练样本变多的到时候，模型学习的东西变多，验证误差开始缓慢的下降。但是一条直线不可能很好的拟合这些数据，因此最后误差会到达在一个高原地带并趋于稳定，最后和训练集的曲线非常接近。

上面的曲线表现了一个典型的欠拟合模型，两条曲线都到达高原地带并趋于稳定，并且最后两条曲线非常接近，同时误差值非常大。

> 提示
>
> 如果你的模型在训练集上是欠拟合的，添加更多的样本是没用的。你需要使用一个更复杂的模型或者找到更好的特征。

现在让我们看一个在相同数据上10阶多项式模型拟合的学习曲线（图 4-16）：

```
from sklearn.pipeline import Pipeline

polynomial_regression = Pipeline((
    ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
    ("sgd_reg", LinearRegression()),
))

plot_learning_curves(polynomial_regression, X, y)
```

这幅图像和之前的有一点点像，但是其有两个非常重要的不同点：

- 在训练集上，误差要比线性回归模型低的多。
- 图中的两条曲线之间有间隔，这意味模型在训练集上的表现要比验证集上好的多，这也是模型过拟合的显著特点。当然，如果你使用了更大的训练数据，这两条曲线最后会非

常的接近。

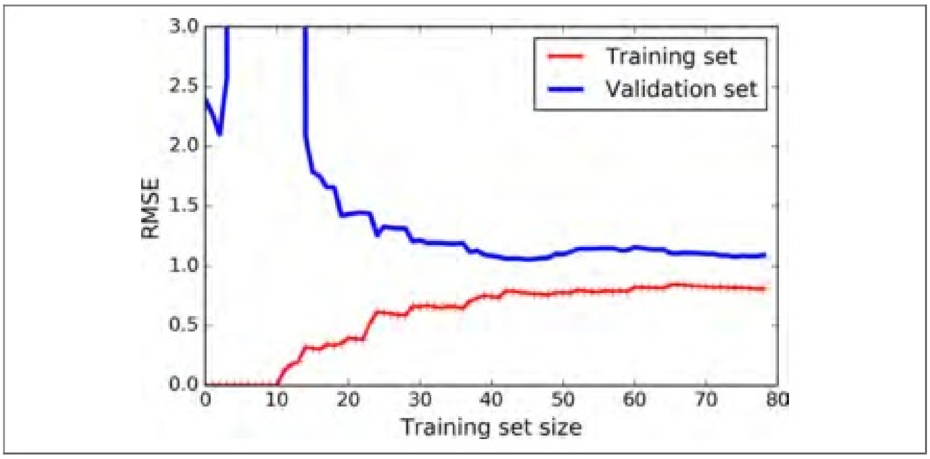


图4-16：多项式模型的学习曲线

> 提示
>
> 改善模型过拟合的一种方法是提供更多的训练数据，直到训练误差和验证误差相等。
>
> 偏差和方差的权衡
>
> 在统计和机器学习领域有个重要的理论：一个模型的泛化误差由三个不同误差的和决定：
>
> - 偏差：泛化误差的这部分误差是由于错误的假设决定的。例如实际是一个二次模型，你却假设了一个线性模型。一个高偏差的模型最容易出现欠拟合。
> - 方差：这部分误差是由于模型对训练数据的微小变化较为敏感，一个多自由度的模型更容易有高的方差（例如一个高阶多项式模型），因此会导致模型过拟合。
> - 不可约误差：这部分误差是由于数据本身的噪声决定的。降低这部分误差的唯一方法就是进行数据清洗（例如：修复数据源，修复坏的传感器，识别和剔除异常值）。

线性模型的正则化

正如我们在第一和第二章看到的那样，降低模型的过拟合的好方法是正则化这个模型（即限制它）：模型有越少的自由度，就越难以拟合数据。例如，正则化一个多项式模型，一个简单的方法就是减少多项式的阶数。

对于一个线性模型，正则化的典型实现就是约束模型中参数的权重。 接下来我们将介绍三种不同约束权重的方法：Ridge 回归，Lasso 回归和 Elastic Net。

岭（Ridge）回归

岭回归（也称为 Tikhonov 正则化）是线性回归的正则化版：在损失函数上直接加上一个正则项 $\alpha\sum_{i=1}^{n}\theta_i^2$ 。这使得学习算法不仅能够拟合数据，而且能够使模型的参数权重尽量的小。注意到这个正则项只有在训练过程中才会被加到损失函数。当得到完成训练的模型后，我们应该使用没有正则化的测量方法去评价模型的表现。

> 提示
>
> 一般情况下，训练过程使用的损失函数和测试过程使用的评价函数是不一样的。除了正则化，还有一个不同：训练时的损失函数应该在优化过程中易于求导，而在测试过程中，评价函数更应该接近最后的客观表现。一个好的例子：在分类训练中我们使用对数损失（马上我们会讨论它）作为损失函数，但是我们却使用精确率/召回率来作为它的评价函数。

超参数 α 决定了你想正则化这个模型的强度。如果 $\alpha=0$ 那此时的岭回归便变为了线性回归。如果 α 非常的大，所有的权重最后都接近于零，最后结果将是一条穿过数据平均值的水平直线。公式 4-8 是岭回归的损失函数：

公式 4-8：岭回归损失函数

$$J(\theta)=MSE(\theta)+\alpha\frac{1}{2}\sum_{i=1}^{n}\theta_i^2$$

值得注意的是偏差 θ_0 是没有被正则化的（累加运算的开始是 $i=1$ 而不是 $i=0$）。如我定义 $\mathbf{w}$ 作为特征的权重向量（θ_1 到 θ_n），那么正则项可以简写成 $\frac{1}{2}(\parallel\mathbf{w}\parallel_2)^2$，其中 $\parallel\cdot\parallel_2$ 表示权重向量的 ℓ_2 范数。对于梯度下降来说仅仅在均方差梯度向量（公式 4-6）加上一项 $\alpha\mathbf{w}$ 。

> 提示
>
> 在使用岭回归前，对数据进行放缩（可以使用 `StandardScaler` ）是非常重要的，算法对于输入特征的数值尺度（scale）非常敏感。大多数的正则化模型都是这样的。

图 4-17 展示了在相同线性数据上使用不同 α 值的岭回归模型最后的表现。左图中，使用简单的岭回归模型，最后得到了线性的预测。右图中的数据首先使用 10 阶的 `PolynomialFearures` 进行扩展，然后使用 `StandardScaler` 进行缩放，最后将岭模型应用在处理过后的特征上。这就是带有岭正则项的多项式回归。注意当α增大的时候，导致预测曲线变得扁平（即少了极端值，多了一般值），这样减少了模型的方差，却增加了模型的偏差。

对线性回归来说，对于岭回归，我们可以使用封闭方程去计算，也可以使用梯度下降去处理。它们的缺点和优点是一样的。公式 4-9 表示封闭方程的解（矩阵 $\mathbf{A}$ 是一个除了左上角有一个 0 的 $n\times n$ 的单位矩，这个 0 代表偏差项。译者注：偏差 θ_0 不被正则化的）。

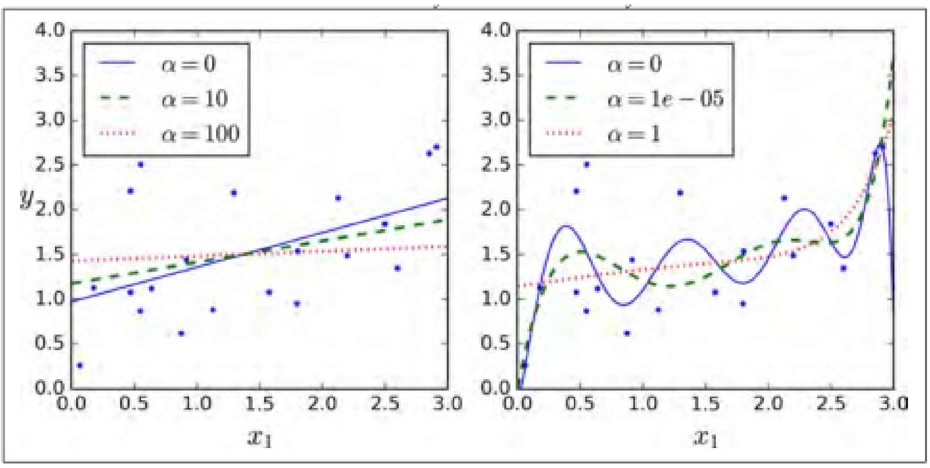


图 4-17：岭回归

公式 4-9：岭回归的封闭方程的解

$$\hat{\theta} = (\mathbf{X}^T \cdot \mathbf{X} + \alpha \mathbf{A})^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

下面是如何使用 Scikit-Learn 来进行封闭方程的求解（使用 Cholesky 法进行矩阵分解对公式 4-9 进行变形）：

```
>>> from sklearn.linear_model import Ridge
>>> ridge_reg = Ridge(alpha=1, solver="cholesky")
>>> ridge_reg.fit(X, y)
>>> ridge_reg.predict([[1.5]])
array([[ 1.55071465]]
```

使用随机梯度法进行求解：

```
>>> sgd_reg = SGDRegressor(penalty="l2")
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.predict([[1.5]])
array([[ 1.13500145]])
```

`penalty` 参数指的是正则项的惩罚类型。指定“l2”表明你要在损失函数上添加一项：权重向量 ℓ_2 范数平方的一半，这就是简单的岭回归。

Lasso 回归

Lasso 回归（也称 Least Absolute Shrinkage，或者 Selection Operator Regression）是另一种正则化版的线性回归：就像岭回归那样，它也在损失函数上添加了一个正则化项，但是它使用权重向量的 ℓ_1 范数而不是权重向量 ℓ_2 范数平方的一半。（如公式 4-10）

公式 4-10：Lasso 回归的损失函数

$$J(\theta) = MSE(\theta) + \alpha \sum_{i=1}^{n} |\theta_i|$$

图 4-18 展示了和图 4-17 相同的事情，仅仅是用 Lasso 模型代替了 Ridge 模型，同时调小了 α 的值。

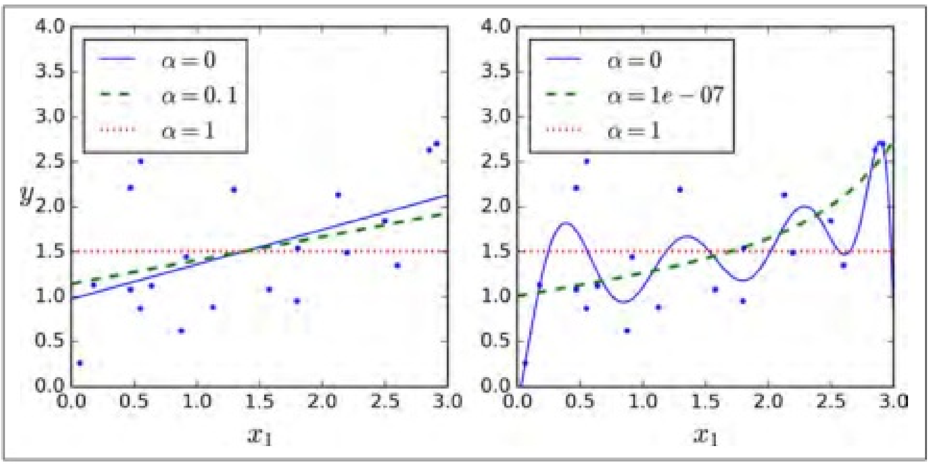


图 4-18：Lasso回归

Lasso 回归的一个重要特征是它倾向于完全消除最不重要的特征的权重（即将它们设置为零）。例如，右图中的虚线所示（$\alpha=10^{-7}$），曲线看起来像一条二次曲线，而且几乎是线性的，这是因为所有的高阶多项特征都被设置为零。换句话说，Lasso回归自动的进行特征选择同时输出一个稀疏模型（即，具有很少的非零权重）。

你可以从图 4-19 知道为什么会出现这种情况：在左上角图中，后背景的等高线（椭圆）表示了没有正则化的均方差损失函数（$\alpha=0$），白色的小圆圈表示在当前损失函数上批量梯度下降的路径。前背景的等高线（菱形）表示ℓ_1惩罚，黄色的三角形表示了仅在这个惩罚下批量梯度下降的路径（$\alpha\to\infty$）。注意路径第一次是如何到达 $\theta_1=0$，然后向下滚动直到它到达 $\theta_2=0$。在右上角图中，等高线表示的是相同损失函数再加上一个 $\alpha=0.5$ 的 ℓ_1 惩罚。这幅图中，它的全局最小值在 $\theta_2=0$ 这根轴上。批量梯度下降首先到达 $\theta_2=0$，然后向下滚动直到达到全局最小值。 两个底部图显示了相同的情况，只是使用了 ℓ_2 惩罚。 规则化的最小值比非规范化的最小值更接近于 $\theta=0$，但权重不能完全消除。

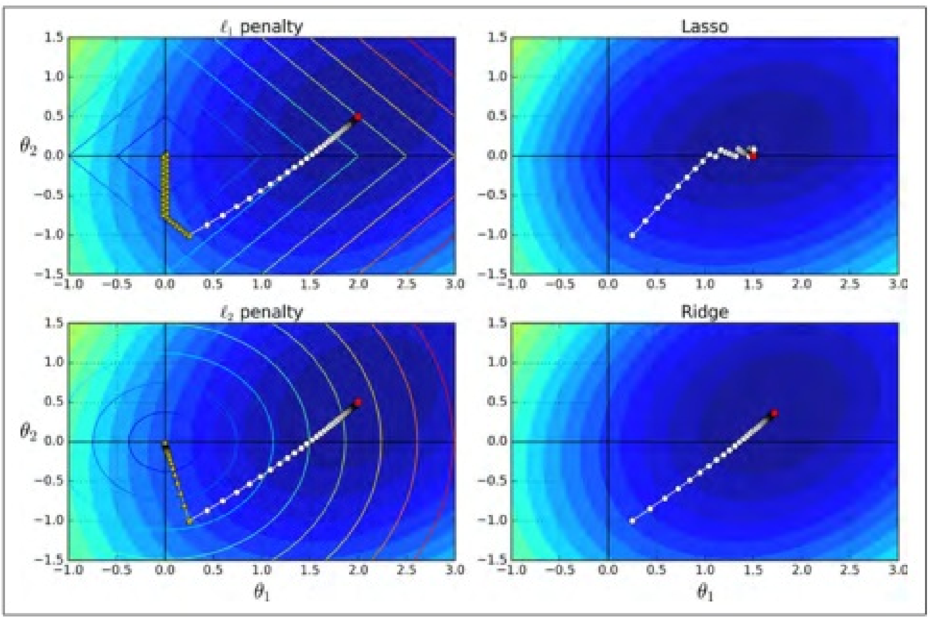


图 4-19：Ridge 回归和 Lasso 回归对比

> 提示
>
> 在 Lasso 损失函数中，批量梯度下降的路径趋向与在低谷有一个反弹。这是因为在 $\theta_2 = 0$ 时斜率会有一个突变。为了最后真正收敛到全局最小值，你需要逐渐的降低学习率。

Lasso 损失函数在 $\theta_i = 0 (i = 1, 2, \cdots, n)$ 处无法进行微分运算，但是梯度下降如果你使用子梯度向量 $\mathbf{g}$ 后它可以在任何 $\theta_i = 0$ 的情况下进行计算。公式 4-11 是在 Lasso 损失函数上进行梯度下降的子梯度向量公式。

公式 4-11：Lasso 回归子梯度向量

$$g(\theta, J) = \nabla_\theta MSE(\theta) + \alpha \begin{pmatrix} sign(\theta_1) \\ sign(\theta_2) \\ \vdots \\ sign(\theta_n) \end{pmatrix} \quad where\ sign(\theta_i) = \begin{cases} -1, & \theta_i < 0 \\ 0, & \theta_i = 0 \\ +1, & \theta_i > 0 \end{cases}$$

下面是一个使用 Scikit-Learn 的 `Lasso` 类的小例子。你也可以使用 `SGDRegressor(penalty="l1")` 来代替它。

```
>>> from sklearn.linear_model import Lasso
>>> lasso_reg = Lasso(alpha=0.1)
>>> lasso_reg.fit(X, y)
>>> lasso_reg.predict([[1.5]])
array([ 1.53788174]
```

弹性网络（ElasticNet）

弹性网络介于 Ridge 回归和 Lasso 回归之间。它的正则项是 Ridge 回归和 Lasso 回归正则项的简单混合，同时你可以控制它们的混合率 r，当 $r = 0$ 时，弹性网络就是 Ridge 回归，当 $r = 1$ 时，其就是 Lasso 回归。具体表示如公式 4-12。

公式 4-12：弹性网络损失函数

$$J(\theta) = MSE(\theta) + r\alpha \sum_{i=1}^{n} |\theta_i| + \frac{1-r}{2} \alpha \sum_{i=1}^{n} \theta_i^2$$

那么我们该如何选择线性回归，岭回归，Lasso 回归，弹性网络呢？一般来说有一点正则项的表现更好，因此通常你应该避免使用简单的线性回归。岭回归是一个很好的首选项，但是如果你的特征仅有少数是真正有用的，你应该选择 Lasso 和弹性网络。就像我们讨论的那样，它两能够将无用特征的权重降为零。一般来说，弹性网络的表现要比 Lasso 好，因为当特征数量比样本的数量大的时候，或者特征之间有很强的相关性时，Lasso 可能会表现的不规律。下面是一个使用 Scikit-Learn `ElasticNet`（`l1_ratio` 指的就是混合率 r）的简单样本：

```
>>> from sklearn.linear_model import ElasticNet
>>> elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
>>> elastic_net.fit(X, y)
>>> elastic_net.predict([[1.5]])
array([ 1.54333232])
```

早期停止法（Early Stopping）

对于迭代学习算法，有一种非常特殊的正则化方法，就像梯度下降在验证错误达到最小值时立即停止训练那样。我们称为早期停止法。图 4-20 表示使用批量梯度下降来训练一个非常复杂的模型（一个高阶多项式回归模型）。随着训练的进行，算法一直学习，它在训练集上的预测误差（RMSE）自然而然的下降。然而一段时间后，验证误差停止下降，并开始上升。这意味着模型在训练集上开始出现过拟合。一旦验证错误达到最小值，便提早停止训练。这种简单有效的正则化方法被 Geoffrey Hinton 称为“完美的免费午餐”

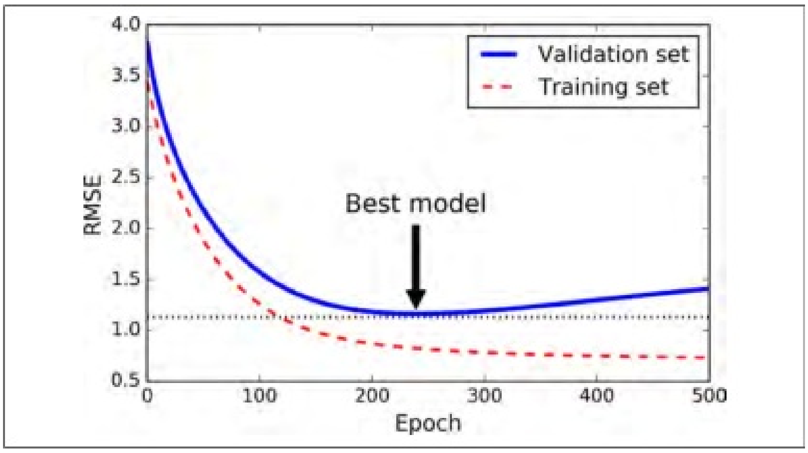


图 4-20：早期停止法

> 提示
>
> 随机梯度和小批量梯度下降不是平滑曲线，你可能很难知道它是否达到最小值。 一种解决方案是，只有在验证误差高于最小值一段时间后（你确信该模型不会变得更好了），才停止，之后将模型参数回滚到验证误差最小值。

下面是一个早期停止法的基础应用：

```
from sklearn.base import clone
sgd_reg = SGDRegressor(n_iter=1, warm_start=True, penalty=None,learning_rate="constant"
, eta0=0.0005)

minimum_val_error = float("inf")
best_epoch = None
best_model = None
for epoch in range(1000):
    sgd_reg.fit(X_train_poly_scaled, y_train)
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    val_error = mean_squared_error(y_val_predict, y_val)
    if val_error < minimum_val_error:
        minimum_val_error = val_error
        best_epoch = epoch
        best_model = clone(sgd_reg)
```

注意：当 `warm_start=True` 时，调用 `fit()` 方法后，训练会从停下来的地方继续，而不是从头重新开始。

逻辑回归

正如我们在第1章中讨论的那样，一些回归算法也可以用于分类（反之亦然）。Logistic 回归（也称为 Logit 回归）通常用于估计一个实例属于某个特定类别的概率（例如，这电子邮件是垃圾邮件的概率是多少？）。如果估计的概率大于 50%，那么模型预测这个实例属于当前类（称为正类，标记为“1”），反之预测它不属于当前类（即它属于负类 ，标记为“0”）。这样便成为了一个二元分类器。

概率估计

那么它是怎样工作的？就像线性回归模型一样，Logistic 回归模型计算输入特征的加权和（加上偏差项），但它不像线性回归模型那样直接输出结果，而是把结果输入 `logistic()` 函数进行二次加工后进行输出（详见公式 4-13）。

公式 4-13：逻辑回归模型的概率估计（向量形式）

$$\hat{p} = h_\theta(\mathbf{x}) = \sigma(\theta^T \cdot \mathbf{x})$$

Logistic 函数（也称为 logit），用 $\sigma()$ 表示，其是一个 sigmoid 函数（图像呈 S 型），它的输出是一个介于 0 和 1 之间的数字。其定义如公式 4-14 和图 4-21 所示。

公式 4-14：逻辑函数

$$\sigma(t) = \frac{1}{1 + exp(-t)}$$

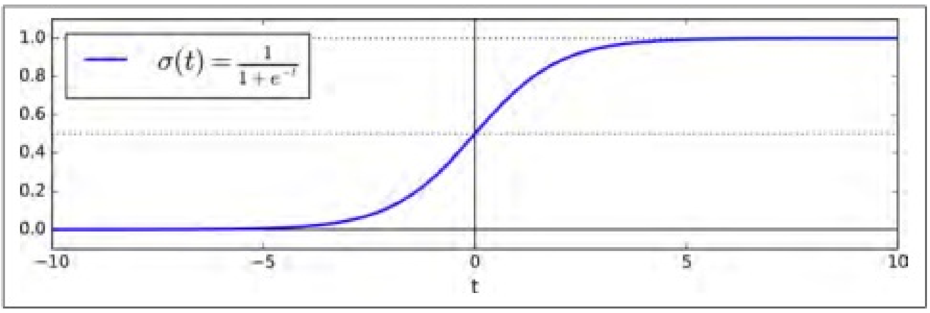


图4-21：逻辑函数

一旦 Logistic 回归模型估计得到了 $\mathbf{x}$ 属于正类的概率 $\hat{p} = h_\theta(\mathbf{x})$，那它很容易得到预测结果 $\hat{y}$（见公式 4-15）。

公式 4-15：逻辑回归预测模型

$$\hat{y} = \begin{cases} 0, & \hat{p} < 0.5 \\ 1, & \hat{p} \geq 0.5 \end{cases}$$

注意当 $t < 0$ 时 $\sigma(t) < 0.5$，当 $t \geq 0$时$\sigma(t) \geq 0.5$，因此当$\theta^T \cdot \mathbf{x}$ 是正数的话，逻辑回归模型输出 1，如果它是负数的话，则输出 0。

训练和损失函数

好，现在你知道了 Logistic 回归模型如何估计概率并进行预测。 但是它是如何训练的？ 训练的目的是设置参数向量 θ，使得正例（$y = 1$）概率增大，负例（$y = 0$）的概率减小，其通过在单个训练实例 $\mathbf{x}$ 的损失函数来实现（公式 4-16）。

公式 4-16：单个样本的损失函数

$$c(\theta) = \begin{cases} -log(\hat{p}), & y = 1 \\ -log(1 - \hat{p}), & y = 0 \end{cases}$$

这个损失函数是合理的，因为当 t 接近 0 时，$-log(t)$ 变得非常大，所以如果模型估计一个正例概率接近于 0，那么损失函数将会很大，同时如果模型估计一个负例的概率接近 1，那么损失函数同样会很大。 另一方面，当 t 接近于 1 时，$-log(t)$ 接近 0，所以如果模型估计一个正例概率接近于 0，那么损失函数接近于 0，同时如果模型估计一个负例的概率接近 0，那么损失函数同样会接近于 0， 这正是我们想的。

整个训练集的损失函数只是所有训练实例的平均值。可以用一个表达式（你可以很容易证明）来统一表示，称为对数损失，如公式 4-17 所示。

公式 4-17：逻辑回归的损失函数（对数损失）

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} log\left(\hat{p}^{(i)}\right) + \left(1 - y^{(i)}\right) log\left(1 - \hat{p}^{(i)}\right)\right]$$

但是这个损失函数对于求解最小化损失函数的 θ 是没有公式解的（没有等价的正态方程）。但好消息是，这个损失函数是凸的，所以梯度下降（或任何其他优化算法）一定能够找到全局最小值（如果学习速率不是太大，并且你等待足够长的时间）。公式 4-18 给出了损失函数关于第 j 个模型参数 θ_j 的偏导数。

公式 4-18：逻辑回归损失函数的偏导数

$$\frac{\partial}{\partial\theta_j}J(\theta_j) = \frac{1}{m}\sum_{i=1}^{m}\left(\sigma\left(\theta^T \cdot \mathbf{x}^{(i)}\right) - y^{(i)}\right){x_j}^{(i)}$$

这个公式看起来非常像公式 4-5：首先计算每个样本的预测误差，然后误差项乘以第 j 项特征值，最后求出所有训练样本的平均值。 一旦你有了包含所有的偏导数的梯度向量，你便可以在梯度向量上使用批量梯度下降算法。 也就是说：你已经知道如何训练 Logistic 回归模型。对于随机梯度下降，你当然只需要每一次使用一个实例，对于小批量梯度下降，你将每一次使用一个小型实例集。

决策边界

我们使用鸢尾花数据集来分析 Logistic 回归。 这是一个著名的数据集，其中包含 150 朵三种不同的鸢尾花的萼片和花瓣的长度和宽度。这三种鸢尾花为：Setosa，Versicolor，Virginica（如图 4-22）。



图4-22：三种不同的鸢尾花

让我们尝试建立一个分类器，仅仅使用花瓣的宽度特征来识别 Virginica，首先让我们加载数据：

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> list(iris.keys())
['data', 'target_names', 'feature_names', 'target', 'DESCR']
>>> X = iris["data"][:, 3:] # petal width
>>> y = (iris["target"] == 2).astype(np.int)
```

接下来，我们训练一个逻辑回归模型：

```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression()
log_reg.fit(X, y)
```

我们来看看模型估计的花瓣宽度从 0 到 3 厘米的概率估计（如图 4-23）：

```
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
y_proba = log_reg.predict_proba(X_new)
plt.plot(X_new, y_proba[:, 1], "g-", label="Iris-Virginica")
plt.plot(X_new, y_proba[:, 0], "b--", label="Not Iris-Virginica"
```

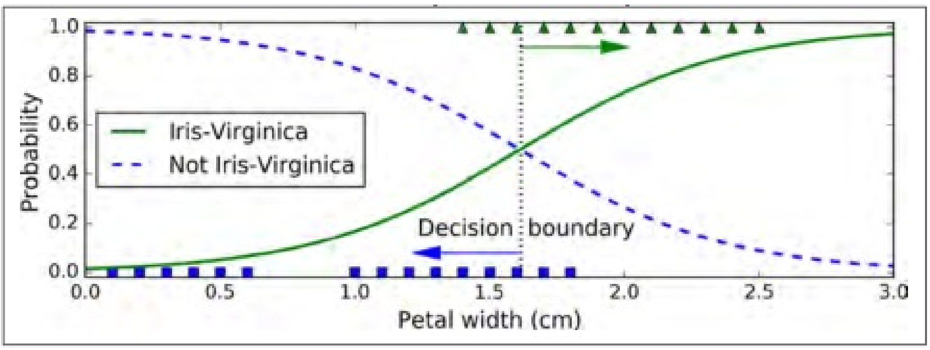


图 4-23：概率估计和决策边界

Virginica 花的花瓣宽度（用三角形表示）在 1.4 厘米到 2.5 厘米之间，而其他种类的花（由正方形表示）通常具有较小的花瓣宽度，范围从 0.1 厘米到 1.8 厘米。注意，它们之间会有一些重叠。在大约 2 厘米以上时，分类器非常肯定这朵花是Virginica花（分类器此时输出一个非常高的概率值），而在1厘米以下时，它非常肯定这朵花不是 Virginica 花（不是 Virginica 花有非常高的概率）。在这两个极端之间，分类器是不确定的。但是，如果你使用它进行预测（使用 `predict()` 方法而不是 `predict_proba()` 方法），它将返回一个最可能的结果。因此，在 1.6 厘米左右存在一个决策边界，这时两类情况出现的概率都等于 50%：如果花瓣宽度大于 1.6 厘米，则分类器将预测该花是 Virginica，否则预测它不是（即使它有可能错了）：

```
>>> log_reg.predict([[1.7], [1.5]])
array([1, 0])
```

图 4-24 表示相同的数据集，但是这次使用了两个特征进行判断：花瓣的宽度和长度。 一旦训练完毕，Logistic 回归分类器就可以根据这两个特征来估计一朵花是 Virginica 的可能性。 虚线表示这时两类情况出现的概率都等于 50%：这是模型的决策边界。 请注意，它是一个线性边界。每条平行线都代表一个分类标准下的两两个不同类的概率，从 15%（左下角）到 90%（右上角）。越过右上角分界线的点都有超过 90% 的概率是 Virginica 花。

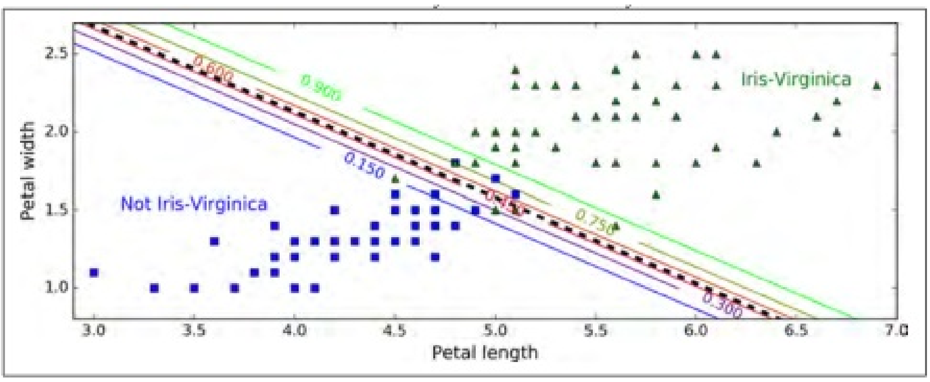


图 4-24：线性决策边界

就像其他线性模型，逻辑回归模型也可以 ℓ_1 或者 ℓ_2 惩罚使用进行正则化。Scikit-Learn 默认添加了 ℓ_2 惩罚。

> 注意
>
> 在 Scikit-Learn 的 `LogisticRegression` 模型中控制正则化强度的超参数不是 α（与其他线性模型一样），而是它的逆：C。 C 的值越大，模型正则化强度越低。

Softmax 回归

Logistic 回归模型可以直接推广到支持多类别分类，不必组合和训练多个二分类器（如第 3 章所述）， 其称为 Softmax 回归或多类别 Logistic 回归。

这个想法很简单：当给定一个实例 $\mathbf{x}$ 时，Softmax 回归模型首先计算 k 类的分数 $s_k(\mathbf{x})$，然后将分数应用在 `Softmax` 函数（也称为归一化指数）上，估计出每类的概率。 计算 $s_k(\mathbf{x})$ 的公式看起来很熟悉，因为它就像线性回归预测的公式一样（见公式 4-19）。

公式 4-19： `k` 类的 Softmax 得分

$$s_k(\mathbf{x}) = \theta^T \cdot \mathbf{x}$$

注意，每个类都有自己独一无二的参数向量 θ_k。 所有这些向量通常作为行放在参数矩阵 Θ 中。

一旦你计算了样本 $\mathbf{x}$ 的每一类的得分，你便可以通过 `Softmax` 函数（公式 4-20）估计出样本属于第 k 类的概率 $\hat{p}_k$：通过计算 e 的 $s_k(\mathbf{x})$ 次方，然后对它们进行归一化（除以所有分子的总和）。

公式 4-20：Softmax 函数

$$\hat{p_k} = \sigma(\mathbf{s}(\mathbf{x}))k = \frac{exp\left(s_k(\mathbf{x})\right)}{\sum_{j=1}^{K} exp\left(s_j\left(\mathbf{x}\right)\right)}$$

- K 表示有多少类
- $\mathbf{s}(\mathbf{x})$ 表示包含样本 $\mathbf{x}$ 每一类得分的向量
- $\sigma(\mathbf{s}(\mathbf{x}))_k$ 表示给定每一类分数之后，实例 $\mathbf{x}$ 属于第 k 类的概率

和 Logistic 回归分类器一样，Softmax 回归分类器将估计概率最高（它只是得分最高的类）的那类作为预测结果，如公式 4-21 所示。

公式 4-21：Softmax 回归模型分类器预测结果

$$\hat{y} = argmax\ \sigma(\mathbf{s}(\mathbf{x}))_k = argmax\ s_k(\mathbf{x}) = argmax\ \left(\theta_k^T \cdot \mathbf{x}\right)$$

- `argmax` 运算返回一个函数取到最大值的变量值。在这个等式，它返回使 $\sigma(\mathbf{s}(\mathbf{x}))_k$ 最大时的 k 的值

> 注意
>
> Softmax 回归分类器一次只能预测一个类（即它是多类的，但不是多输出的），因此它只能用于判断互斥的类别，如不同类型的植物。你不能用它来识别一张照片中的多个人。

现在我们知道这个模型如何估计概率并进行预测，接下来将介绍如何训练。我们的目标是建立一个模型在目标类别上有着较高的概率（因此其他类别的概率较低），最小化公式 4-22 可以达到这个目标，其表示了当前模型的损失函数，称为交叉熵，当模型对目标类得出了一个较低的概率，其会惩罚这个模型。交叉熵通常用于衡量待测类别与目标类别的匹配程度（我们将在后面的章节中多次使用它）

公式 4-22：交叉熵

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} log\left(\hat{p}_k^{(i)}\right)$$

- 如果对于第 i 个实例的目标类是 k，那么 $y_k^{(i)} = 1$，反之 $y_k^{(i)} = 0$。

可以看出，当只有两个类（$K = 2$）时，此损失函数等同于 Logistic 回归的损失函数（对数损失；请参阅公式 4-17）。

> 交叉熵
>
> 交叉熵源于信息论。假设你想要高效地传输每天的天气信息。如果有八个选项（晴天，雨天等），则可以使用3位对每个选项进行编码，因为 $2^3 = 8$。但是，如果你认为几乎每天都是晴天，更高效的编码“晴天”的方式是：只用一位（0）。剩下的七项使用四位（从 1 开始）。交叉熵度量每个选项实际发送的平均比特数。如果你对天气的假设是完美的，交叉熵就等于天气本身的熵（即其内部的不确定性）。但是，如果你的假设是错误的（例如，如果经常下雨）交叉熵将会更大，称为 Kullback-Leibler 散度（KL 散度）。
>
> 两个概率分布 p 和 q 之间的交叉熵定义为：$H(p,q) = -\sum_x p(x)\log q(x)$（分布至少是离散的）

这个损失函数关于 θ_k 的梯度向量为公式 4-23：

公式 4-23：`k` 类交叉熵的梯度向量

$$\nabla_{\theta_k} J(\Theta) = \frac{1}{m} \sum_{i=1}^{m} \left(\hat{p}_k^{(i)} - y_k^{(i)} \right) \mathbf{x}^{(i)}$$

现在你可以计算每一类的梯度向量，然后使用梯度下降（或者其他的优化算法）找到使得损失函数达到最小值的参数矩阵 Θ。

让我们使用 Softmax 回归对三种鸢尾花进行分类。当你使用 `LogisticRregression` 对模型进行训练时，Scikit Learn 默认使用的是一对多模型，但是你可以设置 `multi_class` 参数为“multinomial”来把它改变为 Softmax 回归。你还必须指定一个支持 Softmax 回归的求解器，例如“lbfgs”求解器（有关更多详细信息，请参阅 Scikit-Learn 的文档）。其默认使用 ℓ_12 正则化，你可以使用超参数 C 控制它。

```
X = iris["data"][:, (2, 3)] # petal length, petal width
y = iris["target"]

softmax_reg = LogisticRegression(multi_class="multinomial",solver="lbfgs", C=10)
softmax_reg.fit(X, y)
```

所以下次你发现一个花瓣长为 5 厘米，宽为 2 厘米的鸢尾花时，你可以问你的模型你它是哪一类鸢尾花，它会回答 94.2% 是 Virginica 花（第二类），或者 5.8% 是其他鸢尾花。

```
>>> softmax_reg.predict([[5, 2]])
array([2])
>>> softmax_reg.predict_proba([[5, 2]])
array([[ 6.33134078e-07, 5.75276067e-02, 9.42471760e-01]])是
```

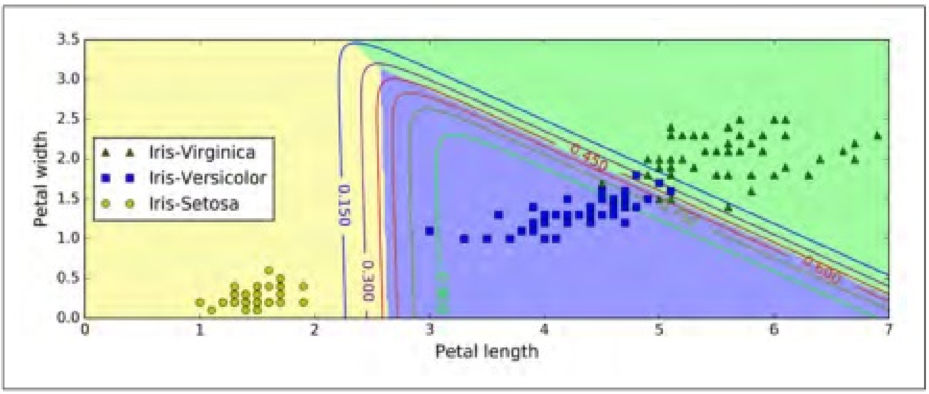


图 4-25：Softmax 回归的决策边界

图 4-25 用不同背景色表示了结果的决策边界。注意，任何两个类之间的决策边界是线性的。该图的曲线表示 Versicolor 类的概率（例如，用 0.450 标记的曲线表示 45% 的概率边界）。注意模型也可以预测一个概率低于 50% 的类。 例如，在所有决策边界相遇的地方，所有类的估计概率相等，分别为 33%。

练习

1. 如果你有一个数百万特征的训练集，你应该选择哪种线性回归训练算法？
2. 假设你训练集中特征的数值尺度（scale）有着非常大的差异，哪种算法会受到影响？有多大的影响？对于这些影响你可以做什么？
3. 训练 Logistic 回归模型时，梯度下降是否会陷入局部最低点？
4. 在有足够的训练时间下，是否所有的梯度下降都会得到相同的模型参数？
5. 假设你使用批量梯度下降法，画出每一代的验证误差。当你发现验证误差一直增大，接下来会发生什么？你怎么解决这个问题？
6. 当验证误差升高时，立即停止小批量梯度下降是否是一个好主意？
7. 哪个梯度下降算法（在我们讨论的那些算法中）可以最快到达解的附近？哪个的确实会收敛？怎么使其他算法也收敛？
8. 假设你使用多项式回归，画出学习曲线，在图上发现学习误差和验证误差之间有着很大的间隙。这表示发生了什么？有哪三种方法可以解决这个问题？
9. 假设你使用岭回归，并发现训练误差和验证误差都很高，并且几乎相等。你的模型表现是高偏差还是高方差？这时你应该增大正则化参数 α，还是降低它？
10. 你为什么要这样做：
    - 使用岭回归代替线性回归？
    - Lasso 回归代替岭回归？
    - 弹性网络代替 Lasso 回归？
11. 假设你想判断一副图片是室内还是室外，白天还是晚上。你应该选择二个逻辑回归分类器，还是一个 Softmax 分类器？
12. 在 Softmax 回归上应用批量梯度下降的早期停止法（不使用 Scikit-Learn）。

附录 A 提供了这些练习的答案。

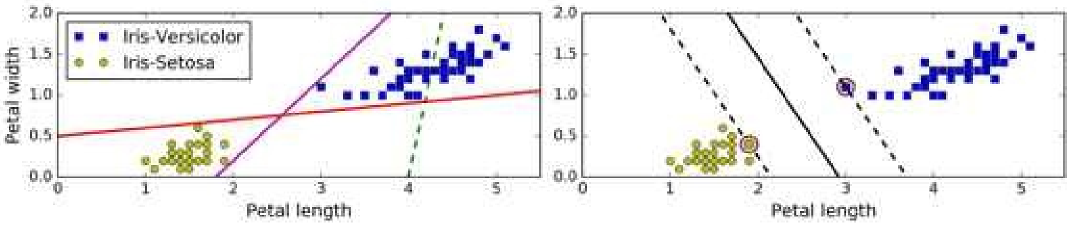
第5章 支持向量机

支持向量机（SVM）是个非常强大并且有多种功能的机器学习模型，能够做线性或者非线性的分类，回归，甚至异常值检测。机器学习领域中最为流行的模型之一，是任何学习机器学习的人必备的工具。SVM 特别适合应用于复杂但中小规模数据集的分类问题。

本章节将阐述支持向量机的核心概念，怎么使用这个强大的模型，以及它是如何工作的。

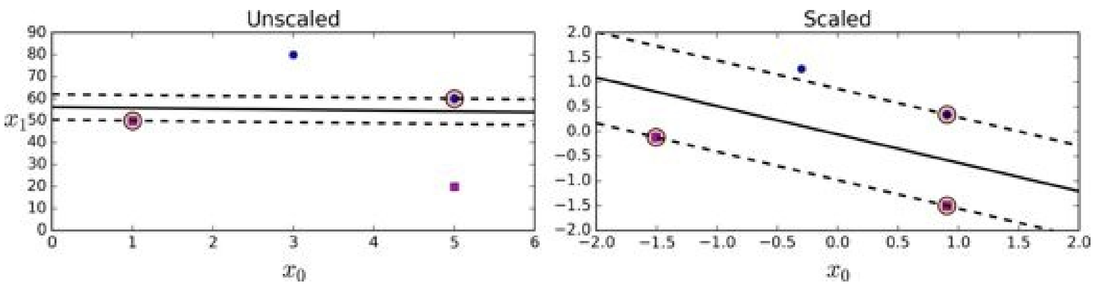
线性支持向量机分类

SVM 的基本思想能够用一些图片来解释得很好，图 5-1 展示了我们在第4章结尾处介绍的鸢尾花数据集的一部分。这两个种类能够被非常清晰，非常容易的用一条直线分开（即线性可分的）。左边的图显示了三种可能的线性分类器的判定边界。其中用虚线表示的线性模型判定边界很差，甚至不能正确地划分类别。另外两个线性模型在这个数据集表现的很好，但是它们的判定边界很靠近样本点，在新的数据上可能不会表现的很好。相比之下，右边图中 SVM 分类器的判定边界实线，不仅分开了两种类别，而且还尽可能地远离了最靠近的训练数据点。你可以认为 SVM 分类器在两种类别之间保持了一条尽可能宽敞的街道（图中平行的虚线），其被称为最大间隔分类。



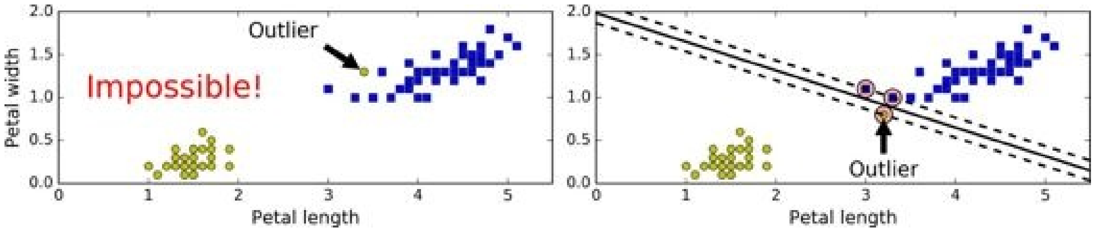
我们注意到添加更多的样本点在“街道”外并不会影响到判定边界，因为判定边界是由位于“街道”边缘的样本点确定的，这些样本点被称为“支持向量”（图 5-1 中被圆圈圈起来的点）

> 警告
>
> SVM 对特征缩放比较敏感，可以看到图 5-2：左边的图中，垂直的比例要更大于水平的比例，所以最宽的“街道”接近水平。但对特征缩放后（例如使用Scikit-Learn的StandardScaler），判定边界看起来要好得多，如右图。



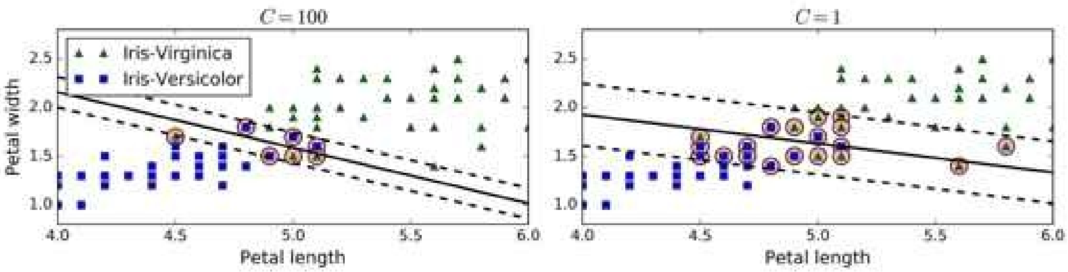
软间隔分类

如果我们严格地规定所有的数据都不在“街道”上，都在正确地两边，称为硬间隔分类，硬间隔分类有两个问题，第一，只对线性可分的数据起作用，第二，对异常点敏感。图 5-3 显示了只有一个异常点的鸢尾花数据集：左边的图中很难找到硬间隔，右边的图中判定边界和我们之前在图 5-1 中没有异常点的判定边界非常不一样，它很难一般化。



为了避免上述的问题，我们更倾向于使用更加软性的模型。目的在保持“街道”尽可能大和避免间隔违规（例如：数据点出现在“街道”中央或者甚至在错误的一边）之间找到一个良好的平衡。这就是软间隔分类。

在 Scikit-Learn 库的 SVM 类，你可以用 `C` 超参数（惩罚系数）来控制这种平衡：较小的 `C` 会导致更宽的“街道”，但更多的间隔违规。图 5-4 显示了在非线性可分隔的数据集上，两个软间隔SVM分类器的判定边界。左边图中，使用了较大的 `C` 值，导致更少的间隔违规，但是间隔较小。右边的图，使用了较小的 `C` 值，间隔变大了，但是许多数据点出现在了“街道”上。然而，第二个分类器似乎泛化地更好：事实上，在这个训练数据集上减少了预测错误，因为实际上大部分的间隔违规点出现在了判定边界正确的一侧。



> 提示
>
> 如果你的 SVM 模型过拟合，你可以尝试通过减小超参数 `C` 去调整。

以下的 Scikit-Learn 代码加载了内置的鸢尾花（Iris）数据集，缩放特征，并训练一个线性 SVM 模型（使用 `LinearSVC` 类，超参数 `C=1`，hinge 损失函数）来检测 Virginica 鸢尾花，生成的模型在图 5-4 的右图。

```
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

iris = datasets.load_iris()
X = iris["data"][:, (2, 3)] # petal length, petal width
y = (iris["target"] == 2).astype(np.float64) # Iris-Virginica

svm_clf = Pipeline((
        ("scaler", StandardScaler()),
        ("linear_svc", LinearSVC(C=1, loss="hinge")),
    ))

svm_clf.fit(X_scaled, y)

Then, as usual, you can use the model to make predictions:

>>> svm_clf.predict([[5.5, 1.7]])
array([ 1.])
```

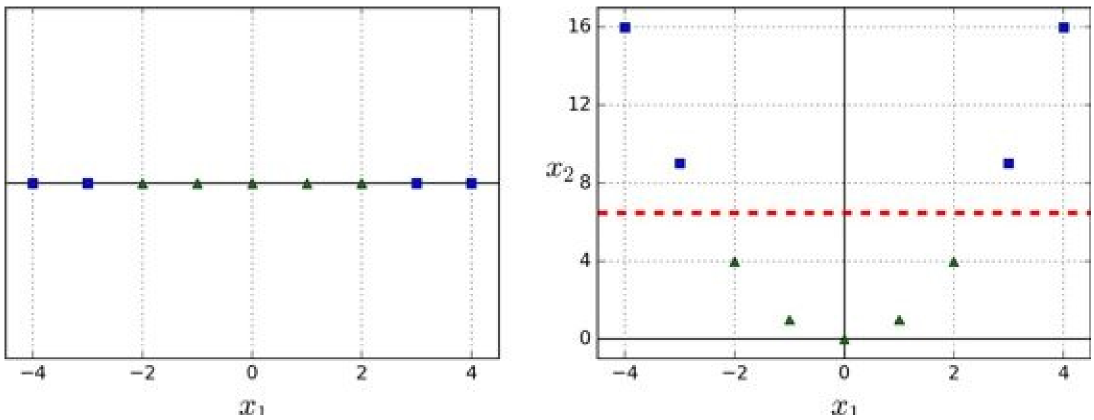
> 注
>
> 不同于 Logistic 回归分类器，SVM 分类器不会输出每个类别的概率。

作为一种选择，你可以在 SVC 类，使用 `SVC(kernel="linear", C=1)` ，但是它比较慢，尤其在较大的训练集上，所以一般不被推荐。另一个选择是使用 `SGDClassifier` 类，即 `SGDClassifier(loss="hinge", alpha=1/(m*C))` 。它应用了随机梯度下降（SGD 见第四章）来训练一个线性 SVM 分类器。尽管它不会和 `LinearSVC` 一样快速收敛，但是对于处理那些不适合放在内存的大数据集是非常有用的，或者处理在线分类任务同样有用。

> 提示
>
> `LinearSVC` 要使偏置项规范化，首先你应该集中训练集减去它的平均数。如果你使用了 `StandardScaler` ，那么它会自动处理。此外，确保你设置 `loss` 参数为 `hinge` ，因为它不是默认值。最后，为了得到更好的效果，你需要将 `dual` 参数设置为 `False` ，除非特征数比样本量多（我们将在本章后面讨论二元性）

非线性支持向量机分类

尽管线性 SVM 分类器在许多案例上表现得出乎意料的好，但是很多数据集并不是线性可分的。一种处理非线性数据集方法是增加更多的特征，例如多项式特征（正如你在第4章所做的那样）；在某些情况下可以变成线性可分的数据。在图 5-5的左图中，它只有一个特征 `x1` 的简单的数据集，正如你看到的，该数据集不是线性可分的。但是如果你增加了第二个特征 `x2=(x1)^2` ，产生的 2D 数据集就能很好的线性可分。

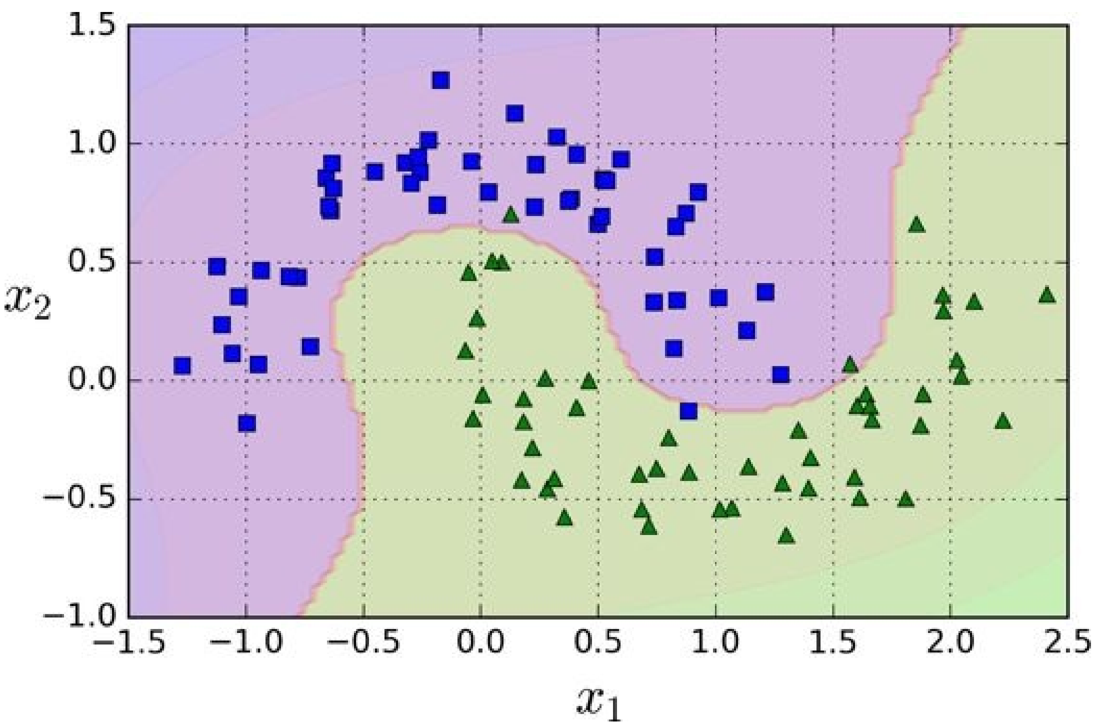


为了实施这个想法，通过 Scikit-Learn，你可以创建一个流水线（Pipeline）去包含多项式特征（PolynomialFeatures）变换（在 121 页的“Polynomial Regression”中讨论），然后一个 `StandardScaler` 和 `LinearSVC` 。让我们在卫星数据集（moons datasets）测试一下效果。

```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

polynomial_svm_clf = Pipeline((
        ("poly_features", PolynomialFeatures(degree=3)),
        ("scaler", StandardScaler()),
        ("svm_clf", LinearSVC(C=10, loss="hinge"))
    ))

polynomial_svm_clf.fit(X, y)
```



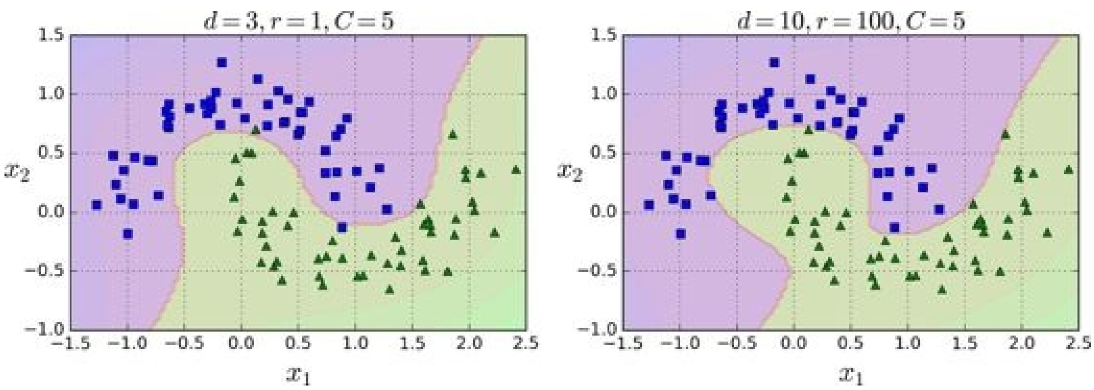
多项式核

添加多项式特征很容易实现，不仅仅在 SVM，在各种机器学习算法都有不错的表现，但是低次数的多项式不能处理非常复杂的数据集，而高次数的多项式却产生了大量的特征，会使模型变得慢。

幸运的是，当你使用 SVM 时，你可以运用一个被称为“核技巧”（kernel trick）的神奇数学技巧。它可以取得就像你添加了许多多项式，甚至有高次数的多项式，一样好的结果。所以不会大量特征导致的组合爆炸，因为你并没有增加任何特征。这个技巧可以用 SVC 类来实现。让我们在卫星数据集测试一下效果。

```
from sklearn.svm import SVC
poly_kernel_svm_clf = Pipeline((
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
    ))
poly_kernel_svm_clf.fit(X, y)
```

这段代码用3阶的多项式核训练了一个 SVM 分类器，即图 5-7 的左图。右图是使用了10阶的多项式核 SVM 分类器。很明显，如果你的模型过拟合，你可以减小多项式核的阶数。相反的，如果是欠拟合，你可以尝试增大它。超参数 `coef0` 控制了高阶多项式与低阶多项式对模型的影响。



通用的方法是用网格搜索（grid search 见第 2 章）去找到最优超参数。首先进行非常粗略的网格搜索一般会很快，然后在找到的最佳值进行更细的网格搜索。对每个超参数的作用有一个很好的理解可以帮助你在正确的超参数空间找到合适的值。

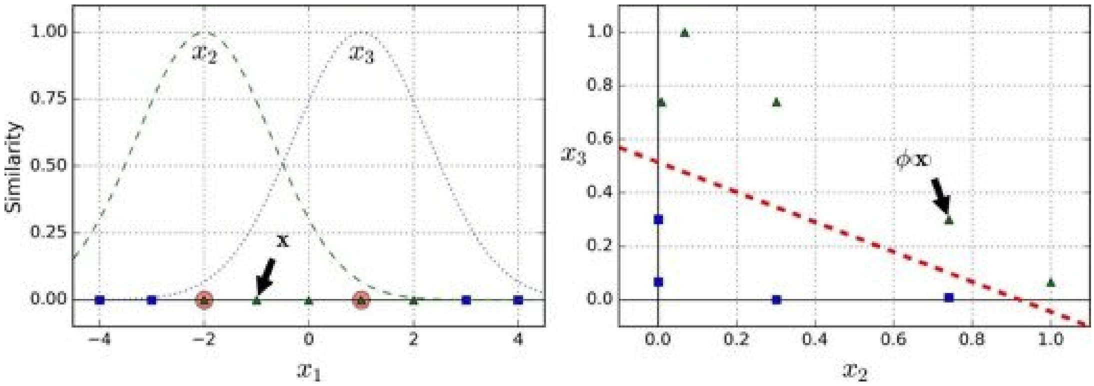
增加相似特征

另一种解决非线性问题的方法是使用相似函数（similarity funtion）计算每个样本与特定地标（landmark）的相似度。例如，让我们来看看前面讨论过的一维数据集，并在 `x1=-2` 和 `x1=1` 之间增加两个地标（图 5-8 左图）。接下来，我们定义一个相似函数，即高斯径向基函数（Gaussian Radial Basis Function，RBF），设置 `γ = 0.3` （见公式 5-1）

公式 5-1 RBF

$$\phi_\gamma(x, \ell) = exp(-\gamma \|x - \ell\|^2)$$

它是个从 0 到 1 的钟型函数，值为 0 的离地标很远，值为 1 的在地标上。现在我们准备计算新特征。例如，我们看一下样本 `x1=-1` ：它距离第一个地标距离是 1，距离第二个地标是 2。因此它的新特征为 `x2=exp(-0.3 × (1^2))≈0.74` 和 `x3=exp(-0.3 × (2^2))≈0.30` 。图 5-8 右边的图显示了特征转换后的数据集（删除了原始特征），正如你看到的，它现在是线性可分了。



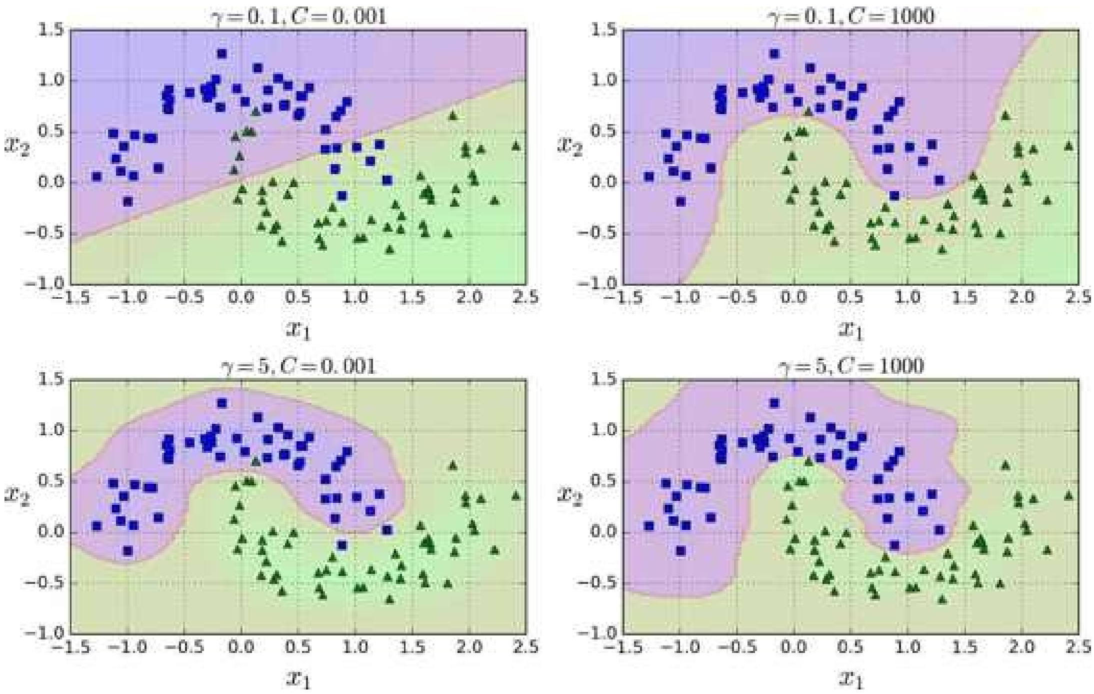
你可能想知道如何选择地标。最简单的方法是在数据集中的每一个样本的位置创建地标。这将产生更多的维度从而增加了转换后数据集是线性可分的可能性。但缺点是， `m` 个样本， `n` 个特征的训练集被转换成了 `m` 个实例， `m` 个特征的训练集（假设你删除了原始特征）。这样一来，如果你的训练集非常大，你最终会得到同样大的特征。

高斯 RBF 核

就像多项式特征法一样，相似特征法对各种机器学习算法同样也有不错的表现。但是在所有额外特征上的计算成本可能很高，特别是在大规模的训练集上。然而，“核” 技巧再一次显现了它在 SVM 上的神奇之处：高斯核让你可以获得同样好的结果成为可能，就像你在相似特征法添加了许多相似特征一样，但事实上，你并不需要在RBF添加它们。我们使用 SVC 类的高斯 RBF 核来检验一下。

```
rbf_kernel_svm_clf = Pipeline((
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))
    ))
rbf_kernel_svm_clf.fit(X, y)
```

这个模型在图 5-9 的左下角表示。其他的图显示了用不同的超参数 `gamma (γ)` 和 `C` 训练的模型。增大 `γ` 使钟型曲线更窄（图 5-8 左图），导致每个样本的影响范围变得更小：即判定边界最终变得更不规则，在单个样本周围环绕。相反的，较小的 `γ` 值使钟型曲线更宽，样本有更大的影响范围，判定边界最终则更加平滑。所以γ是可调整的超参数：如果你的模型过拟合，你应该减小 `γ` 值，若欠拟合，则增大 `γ` （与超参数 `C` 相似）。



还有其他的核函数，但很少使用。例如，一些核函数是专门用于特定的数据结构。在对文本文档或者 DNA 序列进行分类时，有时会使用字符串核（String kernels）（例如，使用 SSK 核（string subsequence kernel）或者基于编辑距离（Levenshtein distance）的核函数）。

> 提示
>
> 这么多可供选择的核函数，你如何决定使用哪一个？一般来说，你应该先尝试线性核函数（记住 `LinearSVC` 比 `SVC(kernel="linear")` 要快得多），尤其是当训练集很大或者有大量的特征的情况下。如果训练集不太大，你也可以尝试高斯径向基核（Gaussian RBF Kernel），它在大多数情况下都很有效。如果你有空闲的时间和计算能力，你还可以使用交叉验证和网格搜索来试验其他的核函数，特别是有专门用于你的训练集数据结构的核函数。

计算复杂性

`LinearSVC` 类基于 `liblinear` 库，它实现了线性 SVM 的优化算法。它并不支持核技巧，但是它样本和特征的数量几乎是线性的：训练时间复杂度大约为 `O(m × n)` 。

如果你要非常高的精度，这个算法需要花费更多时间。这是由容差值超参数 `ϵ` （在 Scikit-learn 称为 `tol` ）控制的。大多数分类任务中，使用默认容差值的效果是已经可以满足一般要求。

SVC 类基于 `libsvm` 库，它实现了支持核技巧的算法。训练时间复杂度通常介 `于O(m^2 × n)` 和 `O(m^3 × n)` 之间。不幸的是，这意味着当训练样本变大时，它将变得极其慢（例如，成千上万个样本）。这个算法对于复杂但小型或中等数量的数据集表现是完美

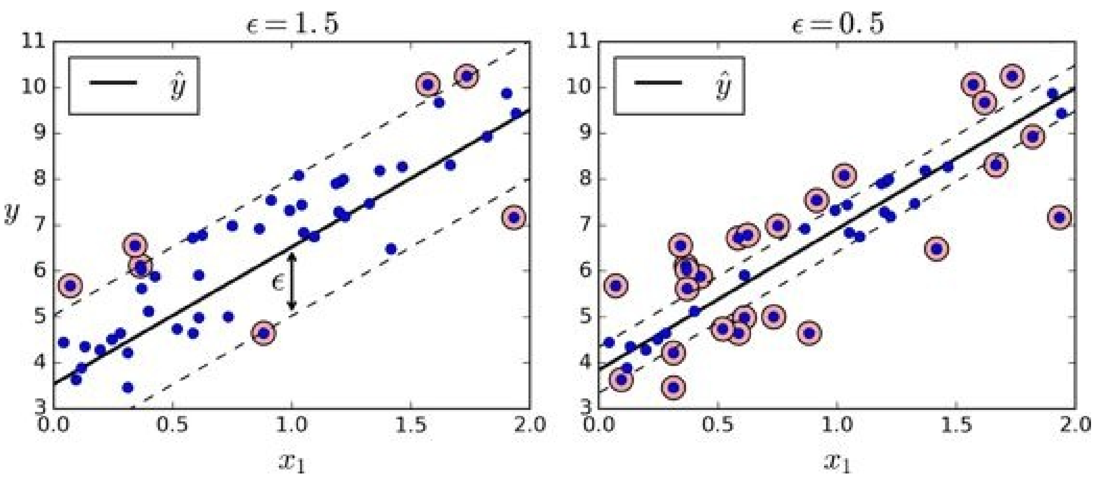
的。然而，它能对特征数量很好的缩放，尤其对稀疏特征来说（sparse features）（即每个样本都有一些非零特征）。在这个情况下，算法对每个样本的非零特征的平均数量进行大概的缩放。表 5-1 对 Scikit-learn 的 SVM 分类模型进行比较。

Table 5-1. Comparison of Scikit-Learn classes for SVM classification

Class	Time complexity	Out-of-core support	Scaling required	Kernel trick
`LinearSVC`	$O(m \times n)$	No	Yes	No
`SGDClassifier`	$O(m \times n)$	Yes	Yes	No
`SVC`	$O(m^2 \times n)$ to $O(m^3 \times n)$	No	Yes	Yes

SVM 回归

正如我们之前提到的，SVM 算法应用广泛：不仅仅支持线性和非线性的分类任务，还支持线性和非线性的回归任务。技巧在于逆转我们的目标：限制间隔违规的情况下，不是试图在两个类别之间找到尽可能大的“街道”（即间隔）。SVM 回归任务是限制间隔违规情况下，尽量放置更多的样本在“街道”上。“街道”的宽度由超参数 `ϵ` 控制。图 5-10 显示了在一些随机生成的线性数据上，两个线性 SVM 回归模型的训练情况。一个有较大的间隔（`ϵ=1.5`），另一个间隔较小（`ϵ=0.5`）。

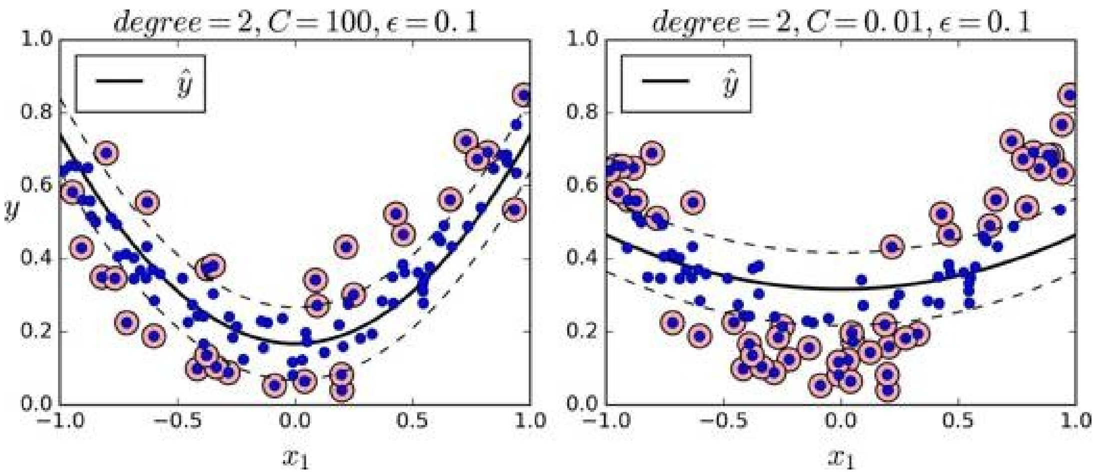


添加更多的数据样本在间隔之内并不会影响模型的预测，因此，这个模型认为是不敏感的（ϵ-insensitive）。

你可以使用 Scikit-Learn 的 `LinearSVR` 类去实现线性 SVM 回归。下面的代码产生的模型在图 5-10 左图（训练数据需要被中心化和标准化）

```
from sklearn.svm import LinearSVR
svm_reg = LinearSVR(epsilon=1.5)
svm_reg.fit(X, y)
```

处理非线性回归任务，你可以使用核化的 SVM 模型。比如，图 5-11 显示了在随机二次方的训练集，使用二次方多项式核函数的 SVM 回归。左图是较小的正则化（即更大的 `C` 值），右图则是更大的正则化（即小的 `C` 值）



下面的代码的模型在图 5-11，其使用了 Scikit-Learn 的 `SVR` 类（支持核技巧）。在回归任务上， `SVR` 类和 `SVC` 类是一样的，并且 `LinearSVR` 是和 `LinearSVC` 等价。 `LinearSVR` 类和训练集的大小成线性（就像 `LinearSVC` 类），当训练集变大， `SVR` 会变的很慢（就像 `SVC` 类）

```
from sklearn.svm import SVR

svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
svm_poly_reg.fit(X, y)
```

> 注
>
> SVM 也可以用来做异常值检测，详情见 Scikit-Learn 文档

背后机制

这个章节从线性 SVM 分类器开始，将解释 SVM 是如何做预测的并且算法是如何工作的。如果你是刚接触机器学习，你可以跳过这个章节，直接进入本章末尾的练习。等到你想深入了解 SVM，再回头研究这部分内容。

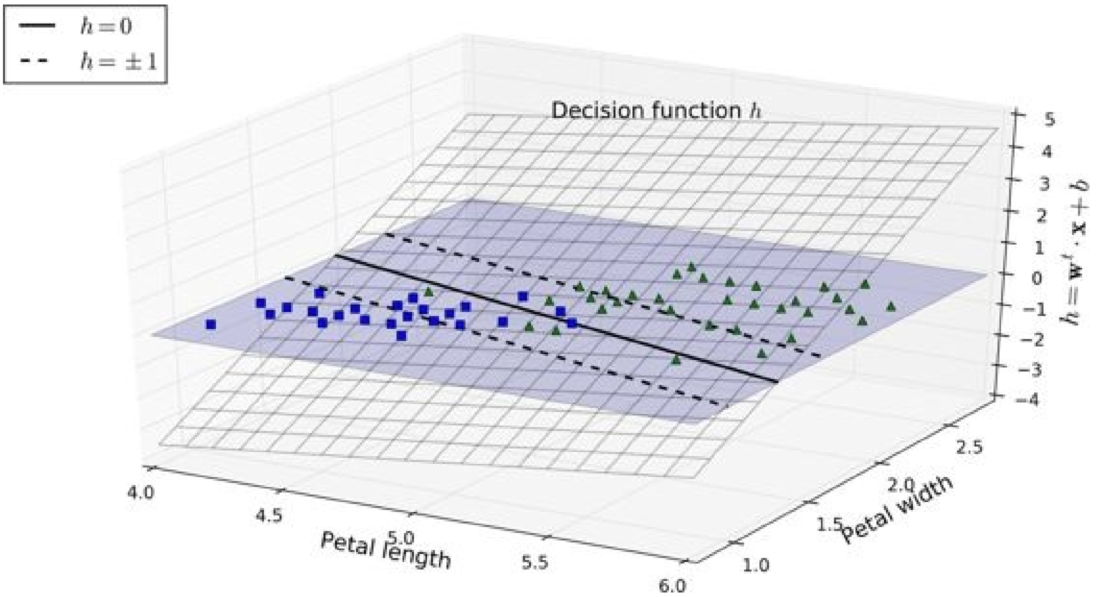
首先，关于符号的约定：在第 4 章，我们将所有模型参数放在一个矢量 `θ` 里，包括偏置项 `θ0` ， `θ1` 到 `θn` 的输入特征权重，和增加一个偏差输入 `x0 = 1` 到所有样本。在本章中，我们将使用一个不同的符号约定，在处理 SVM 上，这更方便，也更常见：偏置项被命名为 `b` ，特征权重向量被称为 `w` ，在输入特征向量中不再添加偏置特征。

决策函数和预测

线性 SVM 分类器通过简单地计算决策函数 $w \cdot x + b = w_1 x_1 + \ldots + w_n x_n + b$ 来预测新样本的类别：如果结果是正的，预测类别 `ŷ` 是正类，为 1，否则他就是负类，为 0。见公式 5-2

$$\hat{y} = \begin{cases} 0 & \text{if } \mathbf{w}^T \cdot \mathbf{x} + b < 0, \\ 1 & \text{if } \mathbf{w}^T \cdot \mathbf{x} + b \geq 0 \end{cases}$$

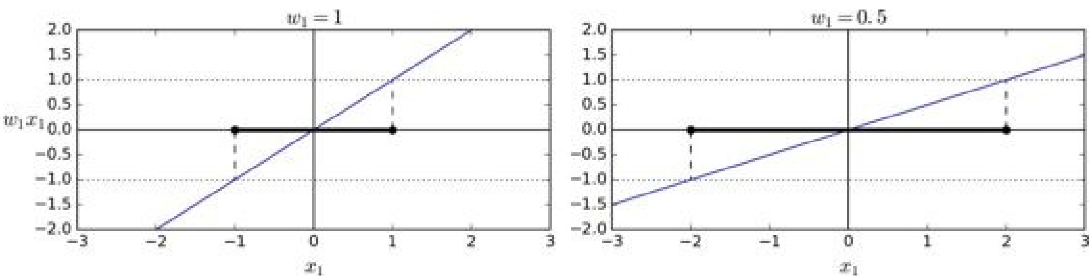
图 5-12 显示了和图 5-4 右边图模型相对应的决策函数：因为这个数据集有两个特征（花瓣的宽度和花瓣的长度），所以是个二维的平面。决策边界是决策函数等于 0 的点的集合，图中两个平面的交叉处，即一条直线（图中的实线）



虚线表示的是那些决策函数等于 1 或 -1 的点：它们平行，且到决策边界的距离相等，形成一个间隔。训练线性 SVM 分类器意味着找到 `w` 值和 `b` 值使得这一个间隔尽可能大，同时避免间隔违规（硬间隔）或限制它们（软间隔）

训练目标

看下决策函数的斜率：它等于权重向量的范数 $\|w\|$。如果我们把这个斜率除于 2，决策函数等于 ±1 的点将会离决策边界原来的两倍大。换句话，即斜率除于 2，那么间隔将增加两倍。在图 5-13 中，2D 形式比较容易可视化。权重向量 `w` 越小，间隔越大。



所以我们的目标是最小化$\|w\|$，从而获得大的间隔。然而，如果我们想要避免间隔违规（硬间隔），对于正的训练样本，我们需要决策函数大于 1，对于负训练样本，小于 -1。若我们对负样本（即 $y^{(i)}=0$）定义 $t^{(i)}=-1$，对正样本（即 $y^{(i)}=1$）定义 $t^{(i)}=1$，那么我们可以对所有的样本表示为 $t^{(i)}(w^Tx^{(i)}+b)\ge 1$。

因此，我们可以将硬间隔线性 SVM 分类器表示为公式 5-3 中的约束优化问题

$$\underset{\mathbf{w},b}{\text{minimize}} \quad \frac{1}{2}\mathbf{w}^T\cdot\mathbf{w}$$
$$\text{subject to} \quad t^{(i)}(\mathbf{w}^T\cdot\mathbf{x}^{(i)}+b)\ge 1 \quad \text{for } i=1,2,\cdots,m$$

> 注
>
> $1/2w^Tw$ 等于 $1/2\|w\|^2$，我们最小化 $1/2w^Tw$，而不是最小化 $\|w\|$。这会给我们相同的结果（因为最小化 `w` 值和 `b` 值，也是最小化该值一半的平方），但是 $1/2\|w\|^2$ 有很好又简单的导数（只有 `w` ），$\|w\|$ 在 `w=0` 处是不可微的。优化算法在可微函数表现得更好。

为了获得软间隔的目标，我们需要对每个样本应用一个松弛变量（slack variable）$\zeta^{(i)}\ge 0$。$\zeta^{(i)}$ 表示了第 `i` 个样本允许违规间隔的程度。我们现在有两个不一致的目标：一个是使松弛变量尽可能的小，从而减小间隔违规，另一个是使 `1/2 w·w` 尽量小，从而增大间隔。这时 `C` 超参数发挥作用：它允许我们在两个目标之间权衡。我们得到了公式 5-4 的约束优化问题。

$$\underset{\mathbf{w},b,\zeta}{\text{minimize}} \quad \frac{1}{2}\mathbf{w}^T\cdot\mathbf{w}+C\sum_{i=1}^{m}\zeta^{(i)}$$
$$\text{subject to} \quad t^{(i)}(\mathbf{w}^T\cdot\mathbf{x}^{(i)}+b)\ge 1-\zeta^{(i)} \quad \text{and} \quad \zeta^{(i)}\ge 0 \quad \text{for } i=1,2,\cdots,m$$

二次规划

硬间隔和软间隔都是线性约束的凸二次规划优化问题。这些问题被称之为二次规划（QP）问题。现在有许多解决方案可以使用各种技术来处理 QP 问题，但这超出了本书的范围。一般问题的公式在公式 5-5 给出。

$$\underset{\mathbf{p}}{\text{Minimize}} \quad \frac{1}{2}\mathbf{p}^T\cdot\mathbf{H}\mathbf{p} \quad + \quad \mathbf{f}^T\cdot\mathbf{p}$$
$$\text{subject to} \quad \mathbf{A}\cdot\mathbf{p}\le\mathbf{b}$$
$$\text{where} \begin{cases} \mathbf{p} & \text{is an } n_p\text{-dimensional vector } (n_p = \text{number of parameters}),\\ \mathbf{H} & \text{is an } n_p\times n_p \text{ matrix},\\ \mathbf{f} & \text{is an } n_p\text{-dimensional vector},\\ \mathbf{A} & \text{is an } n_c\times n_p \text{ matrix } (n_c = \text{number of constraints}),\\ \mathbf{b} & \text{is an } n_c\text{-dimensional vector.}\end{cases}$$

注意到表达式`Ap ≤ b`实际上定义了n_c约束：$p^T a^{(i)} \le b^{(i)}, for i = 1, 2, ..., n$，$a^{(i)}$是个包含了`A`的第`i`行元素的向量，$b^{(i)}$是`b`的第`i`个元素。

可以很容易地看到，如果你用以下的方式设置 QP 的参数，你将获得硬间隔线性 SVM 分类器的目标：

- $n_p = n + 1$，`n`表示特征的数量（+1 是偏置项）
- $n_c = m$，`m`表示训练样本数量
- `H`是$n_p \times n_p$单位矩阵，除了左上角为 0（忽略偏置项）
- `f = 0`，一个全为 0 的n_p维向量
- $b = 1$，一个全为 1 的n_c维向量
- $a^{(i)} = t^{(i)} \dot{x}^{(i)}$，$\dot{x}^{(i)}$等于$x^{(i)}$带一个额外的偏置特征$\dot{x}_0 = 1$！

所以训练硬间隔线性 SVM 分类器的一种方式是使用现有的 QP 解决方案，即上述的参数。由此产生的向量`p`将包含偏置项$b = p_0$和特征权重$w_i = p_i (i = 1, 2, ...m)$。同样的，你可以使用 QP 解决方案来解决软间隔问题（见本章最后的练习）

然而，使用核技巧我们将会看到一个不同的约束优化问题。

对偶问题

给出一个约束优化问题，即原始问题（primal problem），它可能表示不同但是和另一个问题紧密相连，称为对偶问题（Dual Problem）。对偶问题的解通常是对原始问题的解给出一个下界约束，但在某些条件下，它们可以获得相同解。幸运的是，SVM 问题恰好满足这些条件，所以你可以选择解决原始问题或者对偶问题，两者将会有相同解。公式 5-6 表示了线性 SVM 的对偶形式（如果你对怎么从原始问题获得对偶问题感兴趣，可以看下附录 C）

$$\underset{\alpha}{\text{minimize}} \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)T} \cdot \mathbf{x}^{(j)} \quad - \quad \sum_{i=1}^{m} \alpha^{(i)}$$

$$\text{subject to} \quad \alpha^{(i)} \ge 0 \quad \text{for } i = 1, 2, \cdots, m$$

一旦你找到最小化公式的向量`α`（使用 QP 解决方案），你可以通过使用公式 5-7 的方法计算`w`和`b`，从而使原始问题最小化。

$$\hat{\mathbf{w}} = \sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)}$$

$$\hat{b} = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)}>0}}^{m} \left(1 - t^{(i)}\left(\hat{\mathbf{w}}^T \cdot \mathbf{x}^{(i)}\right)\right)$$

当训练样本的数量比特征数量小的时候，对偶问题比原始问题要快得多。更重要的是，它让核技巧成为可能，而原始问题则不然。那么这个核技巧是怎么样的呢？

核化支持向量机

假设你想把一个 2 次多项式变换应用到二维空间的训练集（例如卫星数据集），然后在变换后的训练集上训练一个线性SVM分类器。公式 5-8 显示了你想应用的 2 次多项式映射函数 `ϕ`。

$$\phi(\mathbf{x}) = \phi\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \begin{pmatrix} {x_1}^2 \\ \sqrt{2}\, x_1 x_2 \\ {x_2}^2 \end{pmatrix}$$

注意到转换后的向量是 3 维的而不是 2 维。如果我们应用这个 2 次多项式映射，然后计算转换后向量的点积（见公式 5-9），让我们看下两个 2 维向量 `a` 和 `b` 会发生什么。

$$\phi(\mathbf{a})^T \cdot \phi(\mathbf{b}) = \begin{pmatrix} {a_1}^2 \\ \sqrt{2}\, a_1 a_2 \\ {a_2}^2 \end{pmatrix}^T \cdot \begin{pmatrix} {b_1}^2 \\ \sqrt{2}\, b_1 b_2 \\ {b_2}^2 \end{pmatrix} = {a_1}^2 {b_1}^2 + 2 a_1 b_1 a_2 b_2 + {a_2}^2 {b_2}^2$$

$$= (a_1 b_1 + a_2 b_2)^2 = \left(\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^T \cdot \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \right)^2 = (\mathbf{a}^T \cdot \mathbf{b})^2$$

转换后向量的点积等于原始向量点积的平方：$\phi(a)^T \phi(b) = (a^T b)^2$

关键点是：如果你应用转换 `ϕ` 到所有训练样本，那么对偶问题（见公式 5-6）将会包含点积 $\phi(x^{(i)})^T \phi(x^{(j)})$。但如果 `ϕ` 像在公式 5-8 定义的 2 次多项式转换，那么你可以将这个转换后的向量点积替换成 $(x^{(i)T} x^{(j)})^2$。所以实际上你根本不需要对训练样本进行转换：仅仅需要在公式 5-6 中，将点积替换成它点积的平方。结果将会和你经过麻烦的训练集转换并拟合出线性 SVM 算法得出的结果一样，但是这个技巧使得整个过程在计算上面更有效率。这就是核技巧的精髓。

函数 $K(a, b) = (aTb)^2$ 被称为二次多项式核（polynomial kernel）。在机器学习，核函数是一个能计算点积的函数，并只基于原始向量 `a` 和 `b`，不需要计算（甚至知道）转换 `ϕ`。公式 5-10 列举了一些最常用的核函数。

$$\text{Linear:} \quad K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^T \cdot \mathbf{b}$$

$$\text{Polynomial:} \quad K(\mathbf{a}, \mathbf{b}) = (\gamma \mathbf{a}^T \cdot \mathbf{b} + r)^d$$

$$\text{Gaussian RBF:} \quad K(\mathbf{a}, \mathbf{b}) = \exp(-\gamma \| \mathbf{a} - \mathbf{b} \|^2)$$

$$\text{Sigmoid:} \quad K(\mathbf{a}, \mathbf{b}) = \tanh(\gamma \mathbf{a}^T \cdot \mathbf{b} + r)$$

> Mercer 定理
>
> 根据 Mercer 定理，如果函数 `K(a, b)` 满足一些 Mercer 条件的数学条件(`K` 函数在参数内必须是连续，对称，即 `K(a, b)=K(b, a)` ，等)，那么存在函数 `ϕ` ，将 `a` 和 `b` 映射到另一个空间（可能有更高的维度），有 $K(a, b) = \phi(a)^T \phi(b)$。所以你可以用 `K` 作为核函数，即使你不知道 `ϕ` 是什么。使用高斯核（Gaussian RBF kernel）情况下，它实际是将每个训练样本映射到无限维空间，所以你不需要知道是怎么执行映射的也是一件好事。
>
> 注意一些常用核函数（例如 Sigmoid 核函数）并不满足所有的 Mercer 条件，然而在实践中通常表现得很好。

我们还有一个问题要解决。公式 5-7 展示了线性 SVM 分类器如何从对偶解到原始解，如果你应用了核技巧那么得到的公式会包含 $\phi(x^{(i)})$。事实上，`w` 必须和 $\phi(x^{(i)})$ 有同样的维度，可能是巨大的维度或者无限的维度，所以你很难计算它。但怎么在不知道 `w` 的情况下做出预

测？好消息是你可以将公式 5-7 的 `w` 代入到新的样本 $x^{(n)}$ 的决策函数中，你会得到一个在输入向量之间只有点积的方程式。这时，核技巧将派上用场，见公式 5-11

$$
\begin{aligned}
h_{\mathbf{w},\hat{b}}(\phi(\mathbf{x}^{(n)})) &= \mathbf{w}^T \cdot \phi(\mathbf{x}^{(n)}) + \hat{b} = \Big(\sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} \phi(\mathbf{x}^{(i)})\Big)^T \cdot \phi(\mathbf{x}^{(n)}) + \hat{b} \\
&= \sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} \big(\phi(\mathbf{x}^{(i)})^T \cdot \phi(\mathbf{x}^{(n)})\big) + \hat{b} \\
&= \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)}>0}}^{m} \hat{\alpha}^{(i)} t^{(i)} K(\mathbf{x}^{(i)}, \mathbf{x}^{(n)}) + \hat{b}
\end{aligned}
$$

注意到支持向量才满足`α(i)≠0`，做出预测只涉及计算为支持向量部分的输入样本 $x^{(n)}$ 的点积，而不是全部的训练样本。当然，你同样也需要使用同样的技巧来计算偏置项 `b`，见公式 5-12

$$
\begin{aligned}
\hat{b} &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)}>0}}^{m} \big(1 - t^{(i)} \mathbf{w}^T \cdot \phi(\mathbf{x}^{(i)})\big) = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)}>0}}^{m} \Big(1 - t^{(i)} \big(\sum_{j=1}^{m} \hat{\alpha}^{(j)} t^{(j)} \phi(\mathbf{x}^{(j)})\big)^T \cdot \phi(\mathbf{x}^{(i)})\Big) \\
&= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)}>0}}^{m} \Big(1 - t^{(i)} \sum_{\substack{j=1 \\ \hat{\alpha}^{(j)}>0}}^{m} \hat{\alpha}^{(j)} t^{(j)} K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})\Big)
\end{aligned}
$$

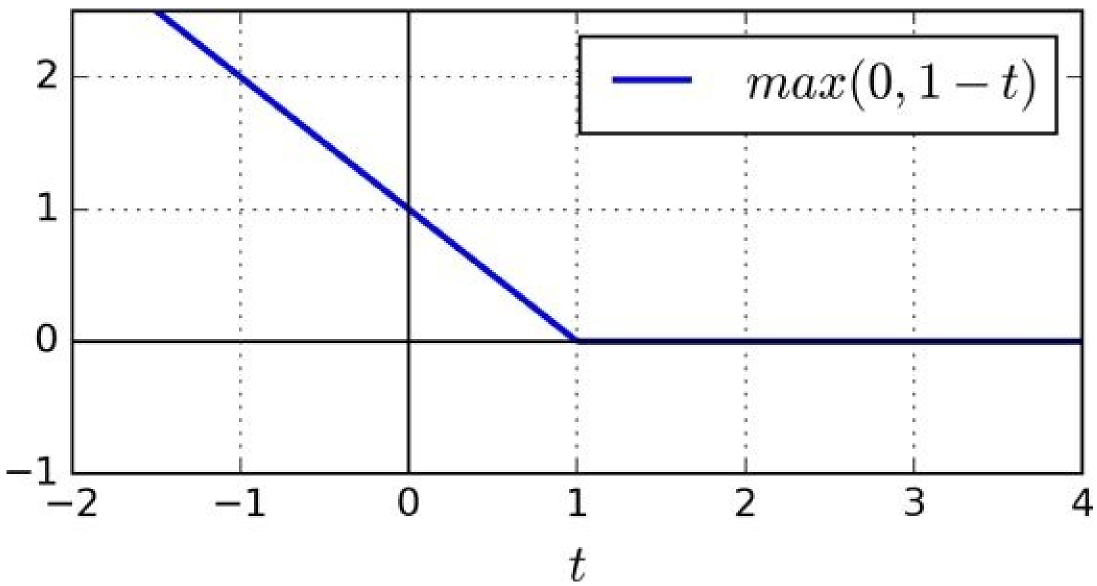
如果你开始感到头痛，这很正常：因为这是核技巧一个不幸的副作用

在线支持向量机

在结束这一章之前，我们快速地了解一下在线 SVM 分类器（回想一下，在线学习意味着增量地学习，不断有新实例）。对于线性SVM分类器，一种方式是使用梯度下降（例如使用`SGDClassifire`）最小化代价函数，如从原始问题推导出的公式 5-13。不幸的是，它比基于 QP 方式收敛慢得多。

$$
J(\mathbf{w}, b) = \frac{1}{2}\mathbf{w}^T \cdot \mathbf{w} \quad + \quad C \sum_{i=1}^{m} max(0,\ 1 - t^{(i)}(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b))
$$

代价函数第一个和会使模型有一个小的权重向量`w`，从而获得一个更大的间隔。第二个和计算所有间隔违规的总数。如果样本位于“街道”上和正确的一边，或它与“街道”正确一边的距离成比例，则间隔违规等于 0。最小化保证了模型的间隔违规尽可能小并且少。

> Hinge 损失
>
> 函数 `max(0, 1-t)` 被称为 Hinge 损失函数（如下）。当 `t≥1` 时，Hinge 值为 0。如果 `t<1`,它的导数（斜率）为 -1，若 `t>1` ，则等于0。在 `t=1` 处，它是不可微的，但就像套索回归（Lasso Regression）（参见 130 页套索回归）一样，你仍然可以在 `t=0` 时使用梯度下降法（即 -1 到 0 之间任何值）
>
> 
>

我们也可以实现在线核化的 SVM。例如使用“增量和递减 SVM 学习”或者“在线和主动的快速核分类器”。但是，这些都是用 Matlab 和 C++ 实现的。对于大规模的非线性问题，你可能需要考虑使用神经网络（见第二部分）

练习

1. 支持向量机背后的基本思想是什么
2. 什么是支持向量
3. 当使用 SVM 时，为什么标准化输入很重要？
4. 分类一个样本时，SVM 分类器能够输出一个置信值吗？概率呢？
5. 在一个有数百万训练样本和数百特征的训练集上，你是否应该使用 SVM 原始形式或对偶形式来训练一个模型？
6. 假设你用 RBF 核来训练一个 SVM 分类器，如果对训练集欠拟合：你应该增大或者减小 `γ` 吗？调整参数 `C` 呢？
7. 使用现有的 QP 解决方案，你应该怎么样设置 QP 参数（ `H` ， `f` ， `A` ，和 `b` ）去解决一个软间隔线性 SVM 分类器问题？
8. 在一个线性可分的数据集训练一个 `LinearSVC` ，并在同一个数据集上训练一个 `SVC` 和 `SGDClassifier` ，看它们是否产生了大致相同效果的模型。

9. 在 MNIST 数据集上训练一个 SVM 分类器。因为 SVM 分类器是二元的分类，你需要使用一对多（one-versus-all）来对 10 个数字进行分类。你可能需要使用小的验证集来调整超参数，以加快进程。最后你能达到多少准确度？

10. 在加利福尼亚住宅（California housing）数据集上训练一个 SVM 回归模型

这些练习的答案在附录 A。

第6章 决策树

和支持向量机一样， 决策树是一种多功能机器学习算法， 即可以执行分类任务也可以执行回归任务， 甚至包括多输出（multioutput）任务.

它是一种功能很强大的算法，可以对很复杂的数据集进行拟合。例如，在第二章中我们对加利福尼亚住房数据集使用决策树回归模型进行训练，就很好的拟合了数据集（实际上是过拟合）。

决策树也是随机森林的基本组成部分（见第 7 章），而随机森林是当今最强大的机器学习算法之一。

在本章中，我们将首先讨论如何使用决策树进行训练，可视化和预测。

然后我们会学习在 Scikit-learn 上面使用 CART 算法，并且探讨如何调整决策树让它可以用于执行回归任务。

最后，我们当然也需要讨论一下决策树目前存在的一些局限性。

决策树的训练和可视化

为了理解决策树，我们需要先构建一个决策树并亲身体验它到底如何进行预测。

接下来的代码就是在我们熟知的鸢尾花数据集上进行一个决策树分类器的训练。

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
iris = load_iris()
X = iris.data[:, 2:] # petal length and width y = iris.target
    tree_clf = DecisionTreeClassifier(max_depth=2)
    tree_clf.fit(X, y)
```

你可以通过使用 `export_graphviz()` 方法，通过生成一个叫做 `iris_tree.dot` 的图形定义文件将一个训练好的决策树模型可视化。

```
from sklearn.tree import export_graphviz
    export_graphviz(
            tree_clf,
            out_file=image_path("iris_tree.dot"),
            feature_names=iris.feature_names[2:],
            class_names=iris.target_names,
            rounded=True,
            filled=True
        )
```

> 译者注：这段代码本人执行不成功， `image_path` 未定义，换其他方法才画出图来。可能是版本原因？

然后，我们可以利用 `graphviz package` [1] 中的 `dot` 命令行，将 `.dot` 文件转换成 PDF 或 PNG 等多种数据格式。例如，使用命令行将 `.dot` 文件转换成 `.png` 文件的命令如下：

> [1] Graphviz是一款开源图形可视化软件包，http://www.graphviz.org/。

```
$ dot -Tpng iris_tree.dot -o iris_tree.png
```

我们的第一个决策树如图 6-1。

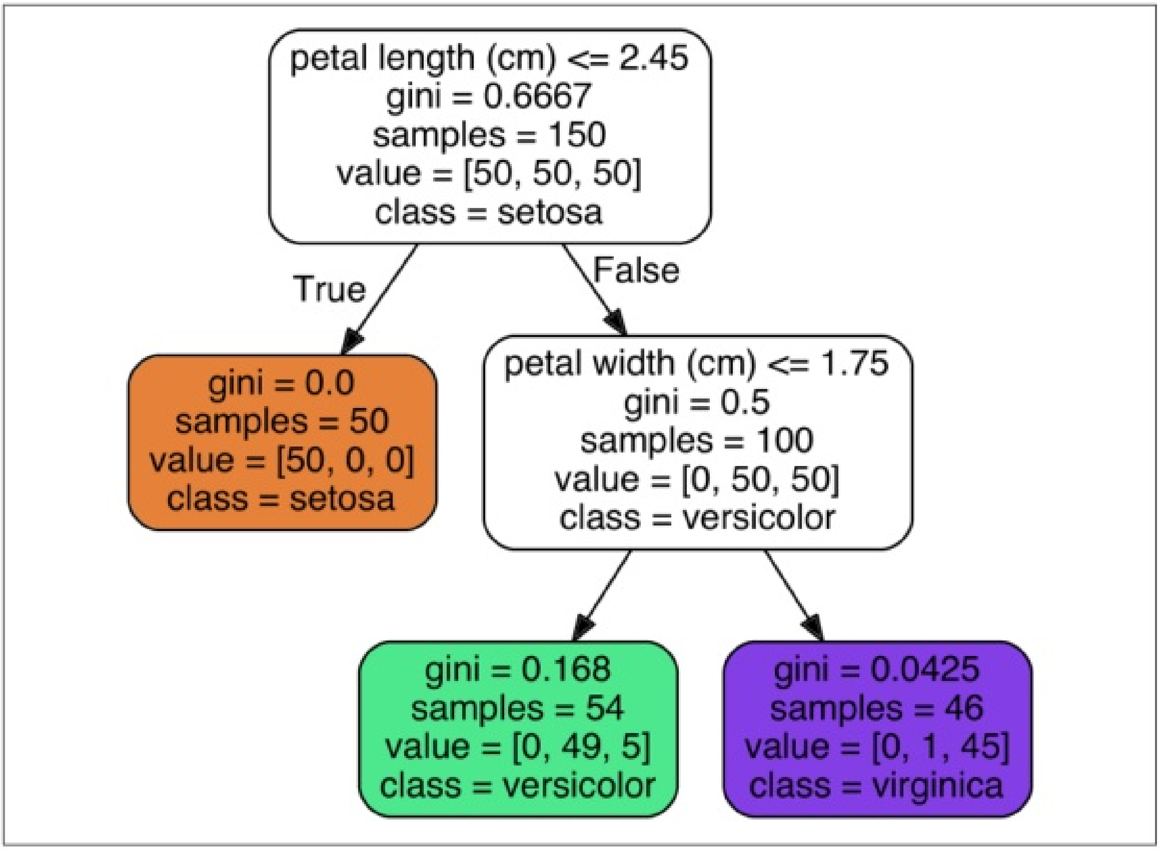


Figure 6-1. Iris Decision Tree

开始预测

现在让我们来看看在图 6-1 中的树是如何进行预测的。假设你找到了一朵鸢尾花并且想对它进行分类，你从根节点开始（深度为 0，顶部）：该节点询问花朵的花瓣长度是否小于 2.45 厘米。如果是，您将向下移动到根的左侧子节点（深度为 1，左侧）。在这种情况下，它是一片叶子节点（即它没有任何子节点），所以它不会问任何问题：你可以方便地查看该节点的预测类别，决策树预测你的花是 Iris-Setosa（`class = setosa`）。

现在假设你找到了另一朵花，但这次的花瓣长度是大于 2.45 厘米的。你必须向下移动到根的右侧子节点（深度为 1，右侧），而这个节点不是叶节点，所以它会问另一个问题：花瓣宽度是否小于 1.75 厘米？如果是，那么你的花很可能是一个 Iris-Versicolor（深度为 2，左）。如果不是，那很可能一个 Iris-Virginica（深度为 2，右），真的是太简单了，对吧！

> 决策树的众多特性之一就是， 它不需要太多的数据预处理， 尤其是不需要进行特征的缩放或者归一化。

节点的 `samples` 属性统计出它应用于多少个训练样本实例。

例如，我们有一百个训练实例是花瓣长度大于 2.45 里面的（深度为 1， 右侧），在这 100 个样例中又有 54 个花瓣宽度小于 1.75cm（深度为 2，左侧）。

节点的 `value` 属性告诉你这个节点对于每一个类别的样例有多少个。

例如：右下角的节点中包含 0 个 Iris-Setosa，1 个 Iris-Versicolor 和 45 个 Iris-Virginica。

最后，节点的 `Gini` 属性用于测量它的纯度：如果一个节点包含的所有训练样例全都是同一类别的，我们就说这个节点是纯的（ `Gini=0` ）。

例如，深度为 1 的左侧节点只包含 Iris-Setosa 训练实例，它就是一个纯节点，Gini 指数为 0。

公式 6-1 显示了训练算法如何计算第 `i` 个节点的 gini 分数 G_i。例如， 深度为 2 的左侧节点基尼指数为：$1-(0/54)^2-(49/54)^2-(5/54)^2=0.68$。另外一个纯度指数也将在后文很快提到。

$$Equation 6-1. Gini impurity$$

$$G_i = 1 - \sum_{k=1}^{n} P_{i,k}^2$$

- $p_{i,k}$ 是第 `i` 个节点中训练实例为的 `k` 类实例的比例

> Scikit-Learn 用的是 CART 算法， CART 算法仅产生二叉树：每一个非叶节点总是只有两个子节点（只有是或否两个结果）。然而，像 ID3 这样的算法可以产生超过两个子节点的决策树模型。

图 6-2 显示了决策树的决策边界。粗的垂直线代表根节点（深度为 0）的决定边界：花瓣长度为 2.45 厘米。由于左侧区域是纯的（只有 Iris-Setosa），所以不能再进一步分裂。然而，右边的区域是不纯的，所以深度为 1 的右边节点在花瓣宽度为 1.75 厘米处分裂（用虚线表示）。又由于 `max_depth` 设置为 2，决策树在那里停了下来。但是，如果将 `max_depth` 设置为 3，两个深度为 2 的节点，每个都将会添加另一个决策边界（用虚线表示）。

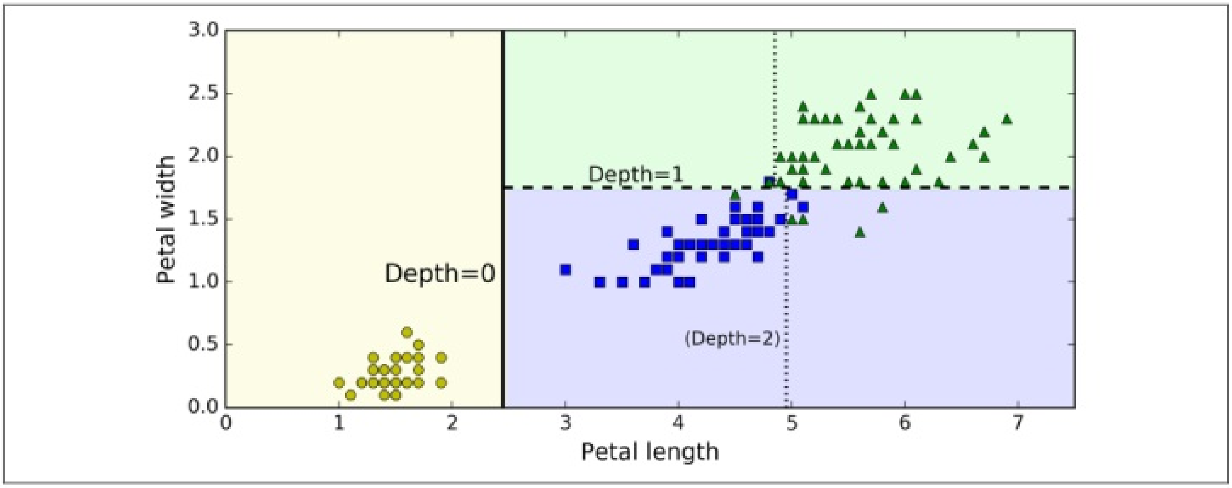


Figure 6-2. Decision Tree decision boundaries

> 模型小知识：白盒与黑盒
>
> 正如我们看到的一样，决策树非常直观，他们的决定很容易被解释。这种模型通常被称为白盒模型。相反，随机森林或神经网络通常被认为是黑盒模型。他们能做出很好的预测，并且您可以轻松检查它们做出这些预测过程中计算的执行过程。然而，人们通常很难用简单的术语来解释为什么模型会做出这样的预测。例如，如果一个神经网络说一个特定的人出现在图片上，我们很难知道究竟是什么导致了这一个预测的出现：
>
> 模型是否认出了那个人的眼睛？ 她的嘴？ 她的鼻子？她的鞋？或者是否坐在沙发上？相反，决策树提供良好的、简单的分类规则，甚至可以根据需要手动操作（例如鸢尾花分类）。

估计分类概率

决策树还可以估计某个实例属于特定类 `k` 的概率：首先遍历树来查找此实例的叶节点，然后它返回此节点中类 `k` 的训练实例的比例。

例如，假设你发现了一个花瓣长 5 厘米，宽 1.5 厘米的花朵。相应的叶节点是深度为 2 的左节点，因此决策树应该输出以下概率：Iris-Setosa 为 0%（0/54），Iris-Versicolor 为 90.7%（49/54），Iris-Virginica 为 9.3%（5/54）。当然，如果你要求它预测具体的类，它应该输出 Iris-Versicolor（类别 1），因为它具有最高的概率。我们了测试一下：

```
>>> tree_clf.predict_proba([[5, 1.5]])
array([[ 0. , 0.90740741, 0.09259259]])
>>> tree_clf.predict([[5, 1.5]])
array([1])
```

完美！请注意，估计概率在任何地方都是相同的， 除了图 6-2 中右下角的矩形部分，例如花瓣长 6 厘米和宽 1.5 厘米（尽管在这种情况下它看起来很可能是 Iris-Virginica）。

CART 训练算法

Scikit-Learn 用分裂回归树（Classification And Regression Tree，简称 CART）算法训练决策树（也叫“增长树”）。这种算法思想真的非常简单：

首先使用单个特征 `k` 和阈值 t_k（例如，“花瓣长度 `≤2.45cm` ”）将训练集分成两个子集。它如何选择 `k` 和 t_k 呢？它寻找到能够产生最纯粹的子集一对 (k, t_k)，然后通过子集大小加权计算。

算法会尝试最小化成本函数。方法如公式 6-2

Equation 6-2. CART cost function for classification

$$J(k,t_k) = \frac{m_{\text{left}}}{m}G_{\text{left}} + \frac{m_{\text{right}}}{m}G_{\text{right}}$$

$$\text{where } \begin{cases} G_{\text{left/right}} \text{ measures the impurity of the left/right subset,} \\ m_{\text{left/right}} \text{ is the number of instances in the left/right subset.} \end{cases}$$

当它成功的将训练集分成两部分之后， 它将会继续使用相同的递归式逻辑继续的分割子集，然后是子集的子集。当达到预定的最大深度之后将会停止分裂（由 `max_depth` 超参数决定），或者是它找不到可以继续降低不纯度的分裂方法的时候。几个其他超参数（之后介绍）控制了其他的停止生长条件（ `min_samples_split` ， `min_samples_leaf` ， `min_weight_fraction_leaf` ， `max_leaf_nodes` ）。

> 正如您所看到的，CART 算法是一种贪婪算法：它贪婪地搜索最高级别的最佳分割方式，然后在每个深度重复该过程。 它不检查分割是否能够在几个级别中的全部分割可能中找到最佳方法。贪婪算法通常会产生一个相当好的解决方法，但它不保证这是全局中的最佳解决方案。

不幸的是，找到最优树是一个 NP 完全问题（自行百度）：它需要 $O(exp^m)$ 时间，即使对于相当小的训练集也会使问题变得棘手。 这就是为什么我们必须设置一个“合理的”（而不是最佳的）解决方案。

计算复杂度

在建立好决策树模型后， 做出预测需要遍历决策树， 从根节点一直到叶节点。决策树通常近似左右平衡，因此遍历决策树需要经历大致 $O(log_2m)$ [2] 个节点。由于每个节点只需要检查一个特征的值，因此总体预测复杂度仅为 $O(log_2m)$，与特征的数量无关。 所以即使在处理大型训练集时，预测速度也非常快。

> [2] log_2 是二进制对数，它等于 $log_2(m) = log(m)/log(2)$。

然而，训练算法的时候（训练和预测不同）需要比较所有特征（如果设置了 `max_features` 会更少一些）

在每个节点的所有样本上。就有了 $O(nmlog(m))$ 的训练复杂度。对于小型训练集（少于几千例），Scikit-Learn 可以通过预先设置数据（ `presort = True` ）来加速训练，但是这对于较大训练集来说会显着减慢训练速度。

基尼不纯度或是信息熵

通常，算法使用 Gini 不纯度来进行检测， 但是你也可以通过将标准超参数设置为 `"entropy"` 来使用熵不纯度进行检测。这里熵的概念是源于热力学中分子混乱程度的概念，当分子井然有序的时候，熵值接近于 0。

熵这个概念后来逐渐被扩展到了各个领域，其中包括香农的信息理论，这个理论被用于测算一段信息中的平均信息密度 [3]。当所有信息相同的时候熵被定义为零。

在机器学习中，熵经常被用作不纯度的衡量方式，当一个集合内只包含一类实例时， 我们称为数据集的熵为 0。

> [3] 熵的减少通常称为信息增益。

公式 6-3 显示了第 `i` 个节点的熵的定义，例如，在图 6-1 中， 深度为 2 左节点的熵为 $-49/54log(49/54)-5/54log(5/54)=0.31$ 。

$$Equation6-3.Entropy$$
$$H_i=-\sum_{k=1 P_{i,k}=!0}^{n} P_{i,k}log(p_i,k)$$

那么我们到底应该使用 Gini 指数还是熵呢？ 事实上大部分情况都没有多大的差别：他们会生成类似的决策树。

基尼指数计算稍微快一点，所以这是一个很好的默认值。但是，也有的时候它们会产生不同的树，基尼指数会趋于在树的分支中将最多的类隔离出来，而熵指数趋向于产生略微平衡一些的决策树模型。

正则化超参数

决策树几乎不对训练数据做任何假设（于此相反的是线性回归等模型，这类模型通常会假设数据是符合线性关系的）。

如果不添加约束，树结构模型通常将根据训练数据调整自己，使自身能够很好的拟合数据，而这种情况下大多数会导致模型过拟合。

这一类的模型通常会被称为非参数模型，这不是因为它没有任何参数（通常也有很多），而是因为在训练之前没有确定参数的具体数量，所以模型结构可以根据数据的特性自由生长。

于此相反的是，像线性回归这样的参数模型有事先设定好的参数数量，所以自由度是受限的，这就减少了过拟合的风险（但是增加了欠拟合的风险）。

`DecisionTreeClassifier` 类还有一些其他的参数用于限制树模型的形状:

> `min_samples_split`（节点在被分裂之前必须具有的最小样本数），`min_samples_leaf`（叶节点必须具有的最小样本数），`min_weight_fraction_leaf`（和`min_samples_leaf`相同，但表示为加权总数的一小部分实例），`max_leaf_nodes`（叶节点的最大数量）`和max_features`（在每个节点被评估是否分裂的时候，具有的最大特征数量）。增加`min_* hyperparameters`或者减少`max_* hyperparameters`会使模型正则化。
>
> 一些其他算法的工作原理是在没有任何约束条件下训练决策树模型，让模型自由生长，然后再对不需要的节点进行剪枝。
>
> 当一个节点的全部子节点都是叶节点时，如果它对纯度的提升不具有统计学意义，我们就认为这个分支是不必要的。
>
> 标准的假设检验，例如卡方检测，通常会被用于评估一个概率值 -- 即改进是否纯粹是偶然性的结果（也叫原假设）
>
> 如果 p 值比给定的阈值更高（通常设定为 5%，也就是 95% 置信度，通过超参数设置），那么节点就被认为是非必要的，它的子节点会被删除。
>
> 这种剪枝方式将会一直进行，直到所有的非必要节点都被删光。

图 6-3 显示了对`moons`数据集（在第 5 章介绍过）进行训练生成的两个决策树模型，左侧的图形对应的决策树使用默认超参数生成（没有限制生长条件），右边的决策树模型设置为`min_samples_leaf=4`。很明显，左边的模型过拟合了，而右边的模型泛用性更好。

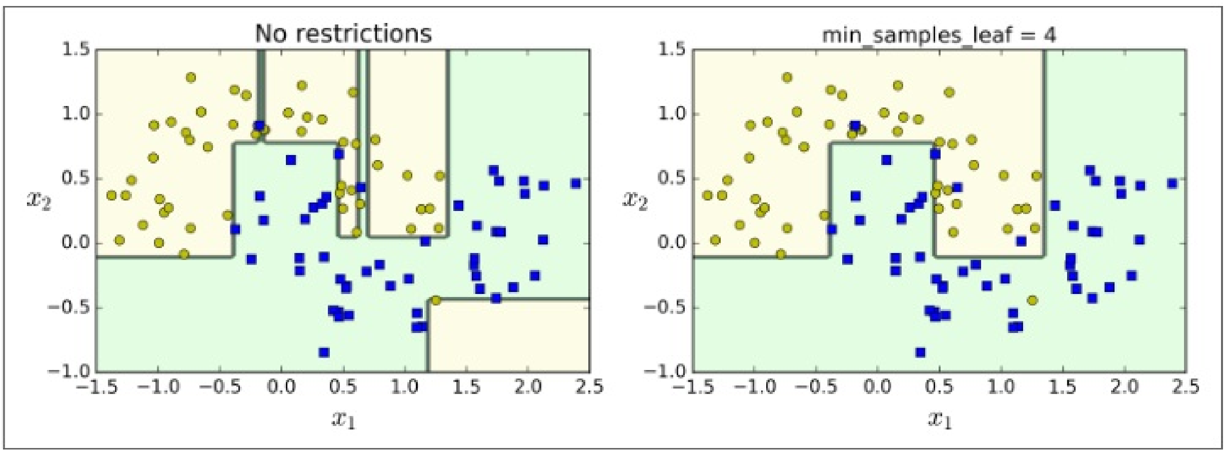


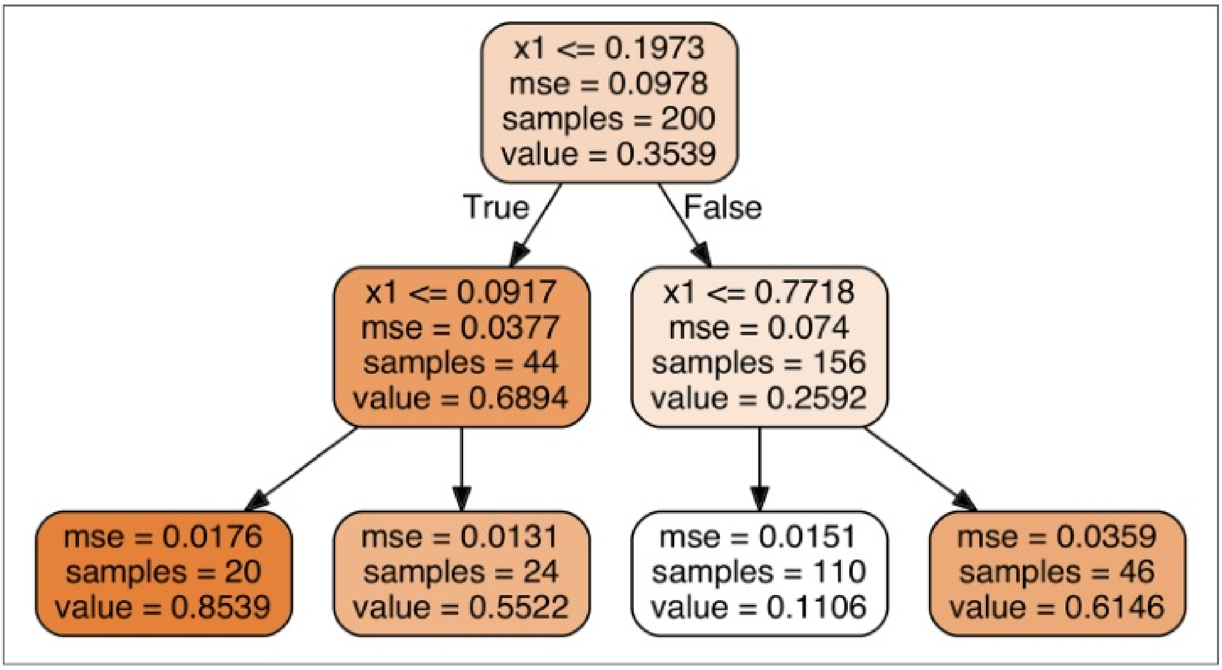
Figure 6-3. Regularization using min_samples_leaf

回归

决策树也能够执行回归任务，让我们使用 Scikit-Learn 的 `DecisionTreeRegressor` 类构建一个回归树，让我们用 `max_depth = 2` 在具有噪声的二次项数据集上进行训练。

```
from sklearn.tree import DecisionTreeRegressor
tree_reg = DecisionTreeRegressor(max_depth=2)
tree_reg.fit(X, y)
```

结果如图 6-4 所示



这棵树看起来非常类似于你之前建立的分类树，它的主要区别在于，它不是预测每个节点中的样本所属的分类，而是预测一个具体的数值。例如，假设您想对 $x_1 = 0.6$ 的新实例进行预测。从根开始遍历树，最终到达预测值等于 0.1106 的叶节点。该预测仅仅是与该叶节点相关的 110 个训练实例的平均目标值。而这个预测结果在对应的 110 个实例上的均方误差（MSE）等于 0.0151。

在图 6-5 的左侧显示的是模型的预测结果，如果你将 `max_depth=3` 设置为 3，模型就会如 6-5 图右侧显示的那样.注意每个区域的预测值总是该区域中实例的平均目标值。算法以一种使大多数训练实例尽可能接近该预测值的方式分割每个区域。

> 译者注：图里面的红线就是训练实例的平均目标值，对应上图中的 `value`

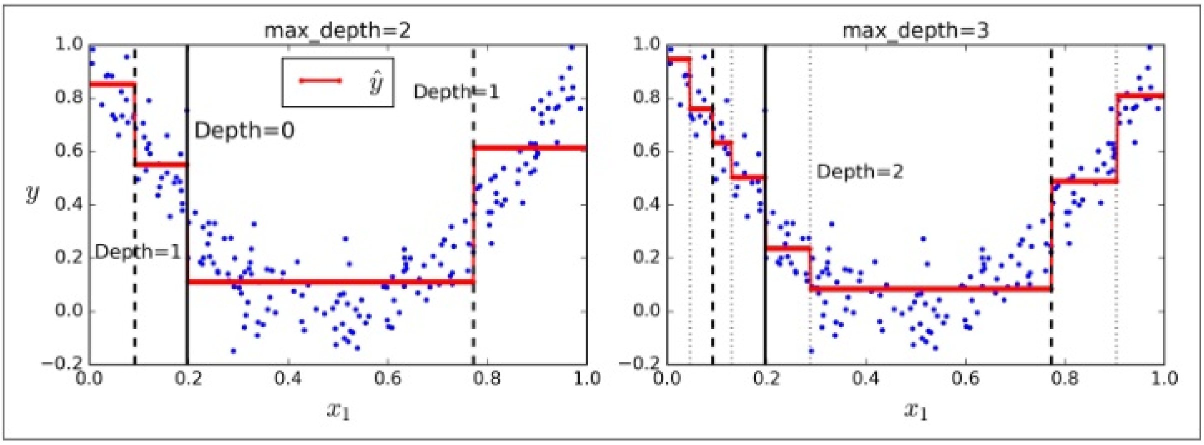


Figure 6-5. Predictions of two Decision Tree regression models

CART 算法的工作方式与之前处理分类模型基本一样，不同之处在于，现在不再以最小化不纯度的方式分割训练集，而是试图以最小化 MSE 的方式分割训练集。

公式 6-4 显示了成本函数，该算法试图最小化这个成本函数。

Equation 6-4. CART cost function for regression

$$J(k, t_k) = \frac{m_{\text{left}}}{m}\text{MSE}_{\text{left}} + \frac{m_{\text{right}}}{m}\text{MSE}_{\text{right}} \quad \text{where} \begin{cases} \text{MSE}_{\text{node}} = \sum_{i \in \text{node}} \left(\hat{y}_{\text{node}} - y^{(i)}\right)^2 \\ \hat{y}_{\text{node}} = \frac{1}{m_{\text{node}}} \sum_{i \in \text{node}} y^{(i)} \end{cases}$$

和处理分类任务时一样，决策树在处理回归问题的时候也容易过拟合。如果不添加任何正则化（默认的超参数），你就会得到图 6-6 左侧的预测结果，显然，过度拟合的程度非常严重。而当我们设置了 `min_samples_leaf = 10`，相对就会产生一个更加合适的模型了，就如图 6-6 所示的那样。

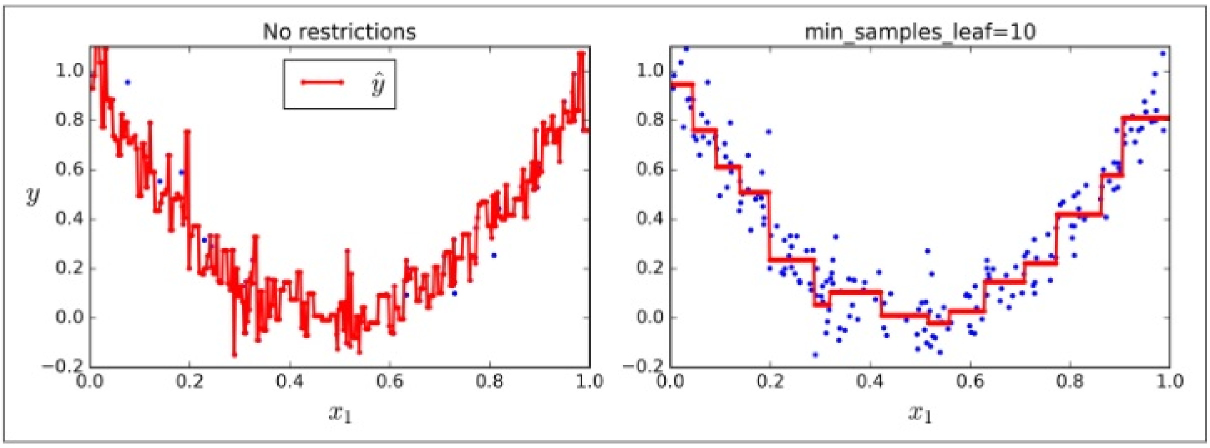


Figure 6-6. Regularizing a Decision Tree regressor

不稳定性

我希望你现在了解了决策树到底有哪些特点：

它很容易理解和解释，易于使用且功能丰富而强大。然而，它也有一些限制，首先，你可能已经注意到了，决策树很喜欢设定正交化的决策边界，（所有边界都是和某一个轴相垂直的），这使得它对训练数据集的旋转很敏感，例如图 6-7 显示了一个简单的线性可分数据集。在左图中，决策树可以轻易的将数据分隔开，但是在右图中，当我们把数据旋转了 45° 之后，决策树的边界看起来变的格外复杂。尽管两个决策树都完美的拟合了训练数据，右边模型的泛化能力很可能非常差。

解决这个难题的一种方式是使用 PCA 主成分分析（第八章），这样通常能使训练结果变得更好一些。

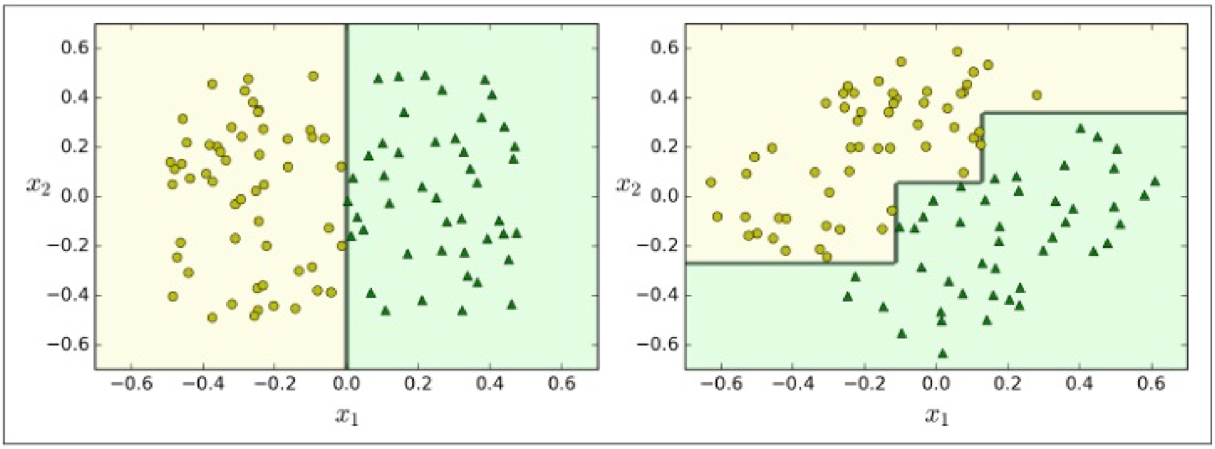


Figure 6-7. Sensitivity to training set rotation

更加通俗的讲，决策时的主要问题是它对训练数据的微小变化非常敏感，举例来说，我们仅仅从鸢尾花训练数据中将最宽的 Iris-Versicolor 拿掉（花瓣长 4.8 厘米，宽 1.8 厘米），然后重新训练决策树模型，你可能就会得到图 6-8 中的模型。正如我们看到的那样，决策树有了非常大的变化（原来的如图 6-2），事实上，由于 Scikit-Learn 的训练算法是非常随机的，即使是相同的训练数据你也可能得到差别很大的模型（除非你设置了随机数种子）。

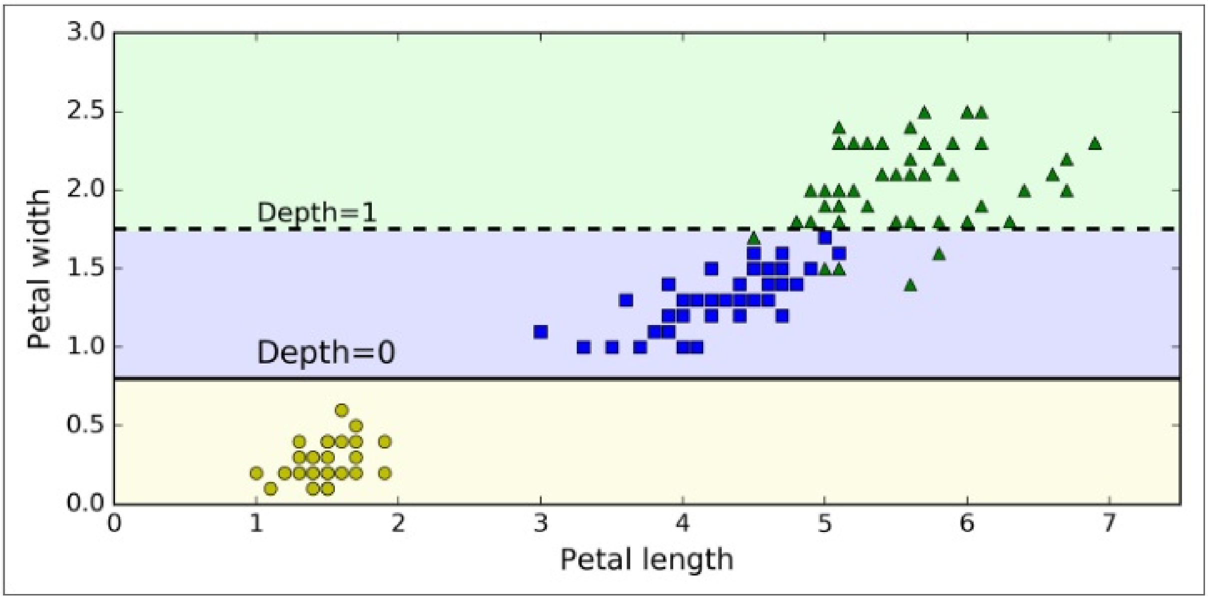


Figure 6-8. Sensitivity to training set details

我们下一章中将会看到，随机森林可以通过多棵树的平均预测值限制这种不稳定性。

练习

1. 在 100 万例训练集上训练（没有限制）的决策树的近似深度是多少？
2. 节点的基尼指数比起它的父节点是更高还是更低？它是通常情况下更高/更低，还是永远更高/更低？
3. 如果决策树过拟合了，减少最大深度是一个好的方法吗？
4. 如果决策树对训练集欠拟合了，尝试缩放输入特征是否是一个好主意？
5. 如果对包含 100 万个实例的数据集训练决策树模型需要一个小时，在包含 1000 万个实例的培训集上训练另一个决策树大概需要多少时间呢？
6. 如果你的训练集包含 100,000 个实例，设置 `presort=True` 会加快训练的速度吗？
7. 对 `moons` 数据集进行决策树训练并优化模型。

    i. 通过语句 `make_moons(n_samples=10000, noise=0.4)` 生成 `moons` 数据集

    ii. 通过 `train_test_split()` 将数据集分割为训练集和测试集。

    iii. 进行交叉验证，并使用网格搜索法寻找最好的超参数值（使用 `GridSearchCV` 类的帮助文档）

    提示: 尝试各种各样的 `max_leaf_nodes` 值

    iv. 使用这些超参数训练全部的训练集数据，并在测试集上测量模型的表现。你应该获得大约 85% 到 87% 的准确度。

8. 生成森林

    i. 接着前边的练习，现在，让我们生成 1,000 个训练集的子集，每个子集包含 100 个随机选择的实例。提示：你可以使用 Scikit-Learn 的 `ShuffleSplit` 类。

    ii. 使用上面找到的最佳超参数值，在每个子集上训练一个决策树。在测试集上测试这 1000 个决策树。由于它们是在较小的集合上进行了训练，因此这些决策树可能会比第一个决策树效果更差，只能达到约 80% 的准确度。

    iii. 见证奇迹的时刻到了！对于每个测试集实例，生成 1,000 个决策树的预测结果，然后只保留出现次数最多的预测结果（您可以使用 SciPy 的 `mode()` 函数）。这个函数使你可以对测试集进行多数投票预测。

    iv. 在测试集上评估这些预测结果，你应该获得了一个比第一个模型高一点的准确率，（大约 0.5% 到 1.5%），恭喜，你已经弄出了一个随机森林分类器模型!

第7章 集成学习和随机森林

假设你去随机问很多人一个很复杂的问题，然后把它们的答案合并起来。通常情况下你会发现这个合并的答案比一个专家的答案要好。这就叫做群体智慧。同样的，如果你合并了一组分类器的预测（像分类或者回归），你也会得到一个比单一分类器更好的预测结果。这一组分类器就叫做集成；因此，这个技术就叫做集成学习，一个集成学习算法就叫做集成方法。

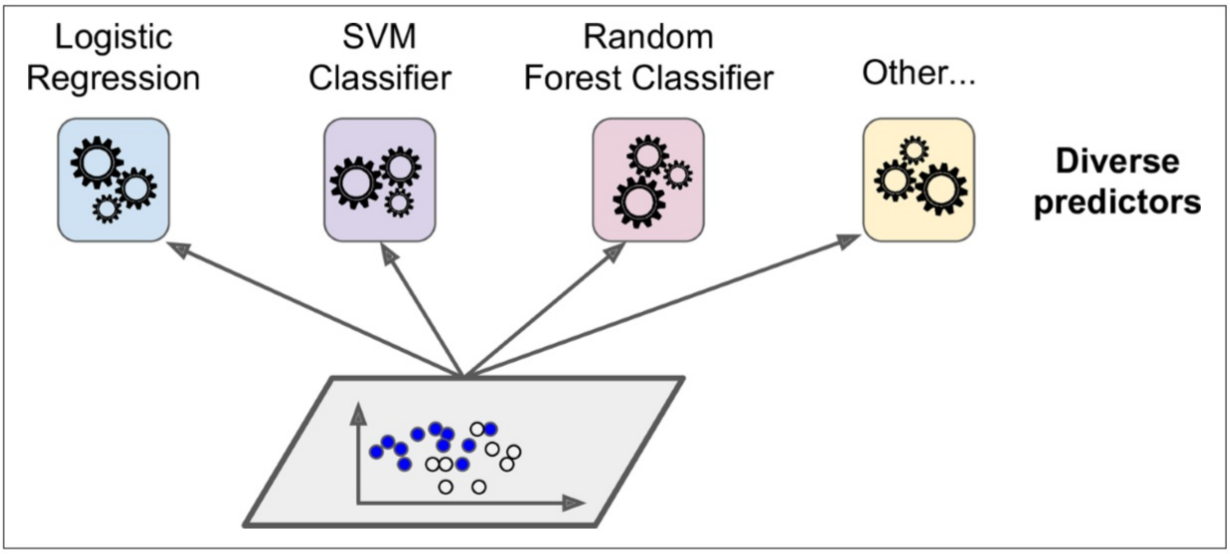
例如，你可以训练一组决策树分类器，每一个都在一个随机的训练集上。为了去做预测，你必须得到所有单一树的预测值，然后通过投票（例如第六章的练习）来预测类别。例如一种决策树的集成就叫做随机森林，它除了简单之外也是现今存在的最强大的机器学习算法之一。

向我们在第二章讨论的一样，我们会在一个项目快结束的时候使用集成算法，一旦你建立了一些好的分类器，就把他们合并为一个更好的分类器。事实上，在机器学习竞赛中获得胜利的算法经常会包含一些集成方法。

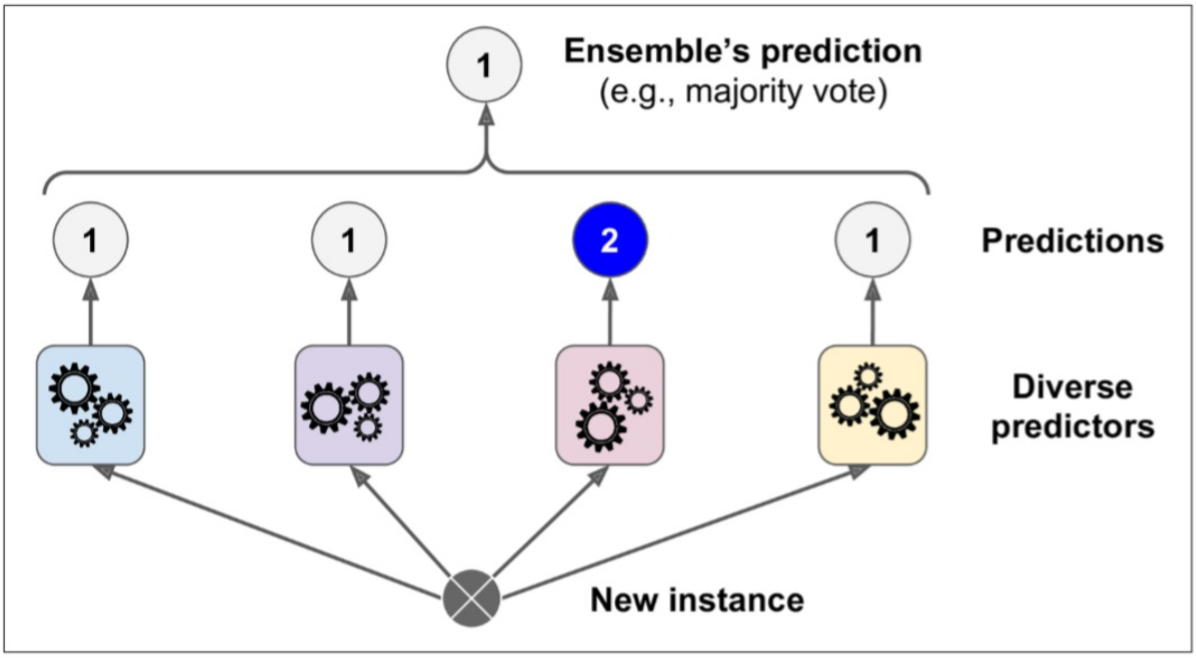
在本章中我们会讨论一下特别著名的集成方法，包括 *bagging, boosting, stacking*，和其他一些算法。我们也会讨论随机森林。

投票分类

假设你已经训练了一些分类器，每一个都有 80% 的准确率。你可能有了一个逻辑斯蒂回归、或一个 SVM、或一个随机森林，或者一个 KNN，或许还有更多（详见图 7-1）

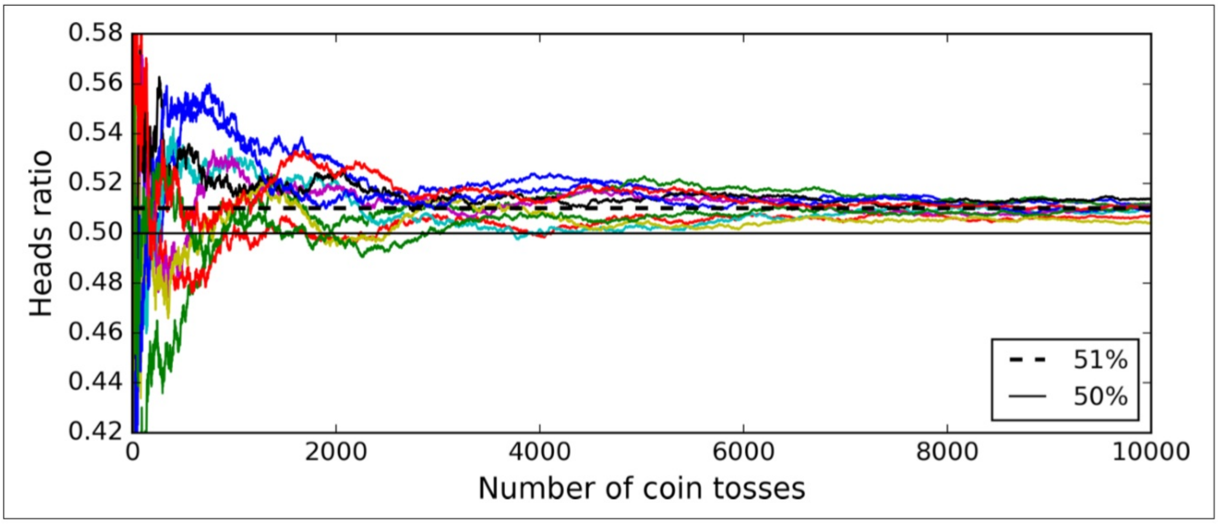


一个非常简单去创建一个更好的分类器的方法就是去整合每一个分类器的预测然后经过投票去预测分类。这种分类器就叫做硬投票分类器（详见图 7-2）。



令人惊奇的是这种投票分类器得出的结果经常会比集成中最好的一个分类器结果更好。事实上，即使每一个分类器都是一个弱学习器（意味着它们也就比瞎猜好点），集成后仍然是一个强学习器（高准确率），只要有足够数量的弱学习者，他们就足够多样化。

这怎么可能？接下来的分析将帮助你解决这个疑问。假设你有一个有偏差的硬币，他有 51% 的几率为正面，49% 的几率为背面。如果你实验 1000 次，你会得到差不多 510 次正面，490 次背面，因此大多数都是正面。如果你用数学计算，你会发现在实验 1000 次后，正面概率为 51% 的人比例为 75%。你实验的次数越多，正面的比例越大（例如你试验了 10000 次，总体比例可能性就会达到 97%）。这是因为大数定律 ：当你一直用硬币实验时，正面的比例会越来越接近 51%。图 7-3 展示了始终有偏差的硬币实验。你可以看到当实验次数上升时，正面的概率接近于 51%。最终所有 10 种实验都会收敛到 51%，它们都大于 50%。



同样的，假设你创建了一个包含 1000 个分类器的集成模型，其中每个分类器的正确率只有 51%（仅比瞎猜好一点点）。如果你用投票去预测类别，你可能得到 75% 的准确率！然而，这仅仅在所有的分类器都独立运行的很好、不会发生有相关性的错误的情况下才会这样，然

而每一个分类器都在同一个数据集上训练，导致其很可能会发生这样的错误。他们可能会犯同一种错误，所以也会有很多票投给了错误类别导致集成的准确率下降。

如果使每一个分类器都独立自主的分类，那么集成模型会工作的很好。去得到多样的分类器的方法之一就是用完全不同的算法，这会使它们会做出不同种类的错误，这会提高集成的正确率

接下来的代码创建和训练了在 sklearn 中的投票分类器。这个分类器由三个不同的分类器组成（训练集是第五章中的 moons 数据集）：

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.ensemble import VotingClassifier
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.svm import SVC
>>> log_clf = LogisticRegression()
>>> rnd_clf = RandomForestClassifier()
>>> svm_clf = SVC()
>>> voting_clf = VotingClassifier(estimators=[('lr', log_clf), ('rf', rnd_clf), >>> ('
svc', svm_clf)],voting='hard')
>>> voting_clf.fit(X_train, y_train)
```

让我们看一下在测试集上的准确率：

```
>>> from sklearn.metrics import accuracy_score
>>> for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
>>>     clf.fit(X_train, y_train)
>>>     y_pred = clf.predict(X_test)
>>>     print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
LogisticRegression 0.864
RandomForestClassifier 0.872
SVC 0.888
VotingClassifier 0.896
```

你看！投票分类器比其他单独的分类器表现的都要好。

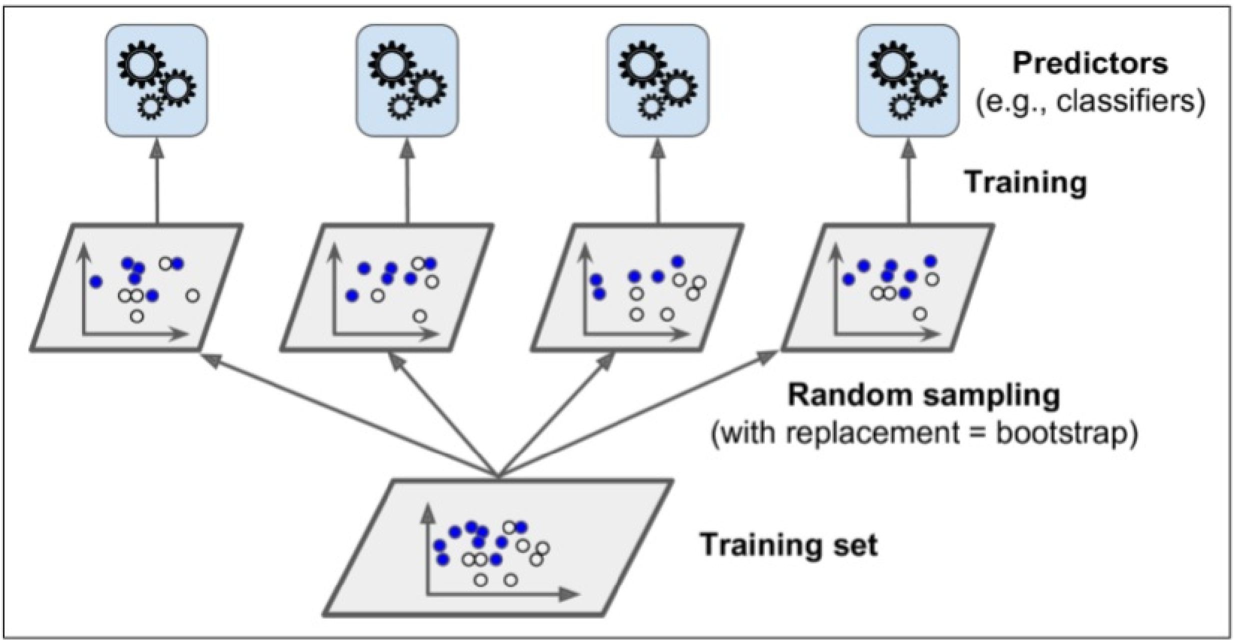
如果所有的分类器都能够预测类别的概率（例如他们有一个 `predict_proba()` 方法），那么你就可以让 sklearn 以最高的类概率来预测这个类，平均在所有的分类器上。这种方式叫做软投票。他经常比硬投票表现的更好，因为它给予高自信的投票更大的权重。你可以通过把 `voting="hard"` 设置为 `voting="soft"` 来保证分类器可以预测类别概率。然而这不是 SVC 类的分类器默认的选项，所以你需要把它的 `probability hyperparameter` 设置为 `True` （这会使 SVC 使用交叉验证去预测类别概率，其降低了训练速度，但会添加 `predict_proba()` 方法）。如果你修改了之前的代码去使用软投票，你会发现投票分类器正确率高达 91%

Bagging 和 Pasting

换句话说，Bagging 和 Pasting 都允许在多个分类器间对训练集进行多次采样，但只有 Bagging

就像之前讲到的，可以通过使用不同的训练算法去得到一些不同的分类器。另一种方法就是对每一个分类器都使用相同的训练算法，但是在不同的训练集上去训练它们。有放回采样被称为装袋（*Bagging*，是 *bootstrap aggregating* 的缩写）。无放回采样称为粘贴（*pasting*）。

换句话说，Bagging 和 Pasting 都允许在多个分类器上对训练集进行多次采样，但只有 Bagging 允许对同一种分类器上对训练集进行进行多次采样。采样和训练过程如图7-4所示。



当所有的分类器被训练后，集成可以通过对所有分类器结果的简单聚合来对新的实例进行预测。聚合函数通常对分类是统计模式（例如硬投票分类器）或者对回归是平均。每一个单独的分类器在如果在原始训练集上都是高偏差，但是聚合降低了偏差和方差。通常情况下，集成的结果是有一个相似的偏差，但是对比与在原始训练集上的单一分类器来讲有更小的方差。

正如你在图 7-4 上所看到的，分类器可以通过不同的 CPU 核或其他的服务器一起被训练。相似的，分类器也可以一起被制作。这就是为什么 Bagging 和 Pasting 是如此流行的原因之一：它们的可扩展性很好。

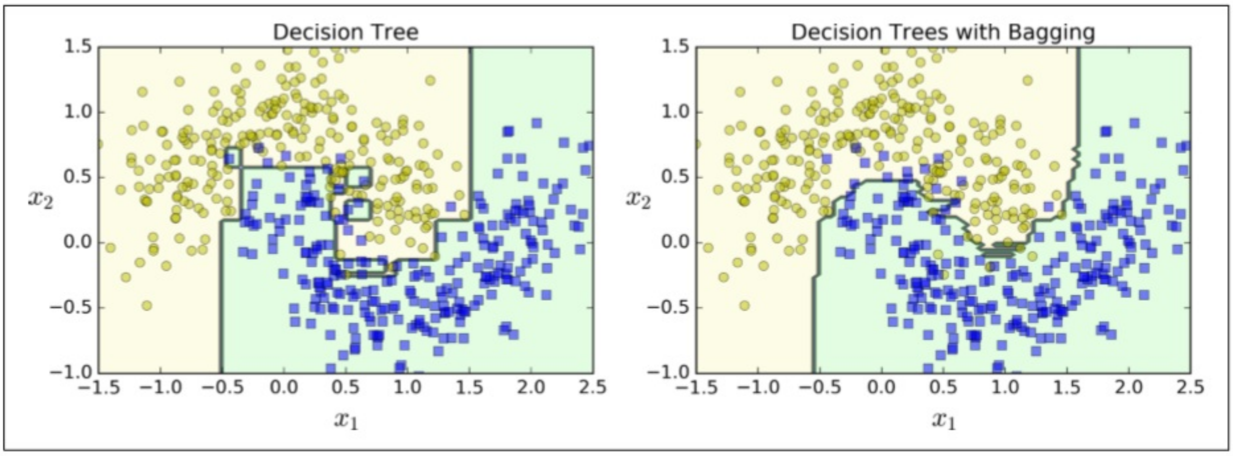
在 sklearn 中的 Bagging 和 Pasting

sklearn 为 Bagging 和 Pasting 提供了一个简单的API：`BaggingClassifier` 类（或者对于回归可以是 `BaggingRegressor` 。接下来的代码训练了一个 500 个决策树分类器的集成，每一个都是在数据集上有放回采样 100 个训练实例下进行训练（这是 Bagging 的例子，如果你想尝试 Pasting，就设置 `bootstrap=False` ）。 `n_jobs` 参数告诉 sklearn 用于训练和预测所需要 CPU 核的数量。（-1 代表着 sklearn 会使用所有空闲核）：

```
>>>from sklearn.ensemble import BaggingClassifier
>>>from sklearn.tree import DecisionTreeClassifier
>>>bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=500,        >>>m
ax_samples=100, bootstrap=True, n_jobs=-1)
>>>bag_clf.fit(X_train, y_train)
>>>y_pred = bag_clf.predict(X_test)
```

如果基分类器可以预测类别概率（例如它拥有 `predict_proba()` 方法），那么 `BaggingClassifier` 会自动的运行软投票，这是决策树分类器的情况。

图 7-5 对比了单一决策树的决策边界和 Bagging 集成 500 个树的决策边界，两者都在 moons 数据集上训练。正如你所看到的，集成的分类比起单一决策树的分类产生情况更好：集成有一个可比较的偏差但是有一个较小的方差（它在训练集上的错误数目大致相同，但决策边界较不规则）。



Bootstrap 在每个预测器被训练的子集中引入了更多的分集，所以 Bagging 结束时的偏差比 Pasting 更高，但这也意味着预测因子最终变得不相关，从而减少了集合的方差。总体而言，Bagging 通常会导致更好的模型，这就解释了为什么它通常是首选的。然而，如果你有空闲时间和 CPU 功率，可以使用交叉验证来评估 Bagging 和 Pasting 哪一个更好。

Out-of-Bag 评价

对于 Bagging 来说，一些实例可能被一些分类器重复采样，但其他的有可能不会被采样。`BaggingClassifier` 默认采样。`BaggingClassifier` 默认是有放回的采样 `m` 个实例（`bootstrap=True`），其中 `m` 是训练集的大小，这意味着平均下来只有63%的训练实例被每个分类器采样，剩下的37%个没有被采样的训练实例就叫做 *Out-of-Bag* 实例。注意对于每一个的分类器它们的 37% 不是相同的。

因为在训练中分类器从来没有看到过 oob 实例，所以它可以在这些实例上进行评估，而不需要单独的验证集或交叉验证。你可以拿出每一个分类器的 oob 来评估集成本身。

在 sklearn 中，你可以在训练后需要创建一个 `BaggingClassifier` 来自动评估时设置 `oob_score=True` 来自动评估。接下来的代码展示了这个操作。评估结果通过变量 `oob_score_` 来显示：

```
>>> bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=500,bootstrap=True, n_jobs=-1, oob_score=True)
>>> bag_clf.fit(X_train, y_train)
>>> bag_clf.oob_score_
0.93066666666666664
```

根据这个 obb 评估，`BaggingClassifier` 可以再测试集上达到93.1%的准确率，让我们修改一下：

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = bag_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred)
0.93600000000000005
```

我们在测试集上得到了 93.6% 的准确率，足够接近了！

对于每个训练实例 oob 决策函数也可通过 `oob_decision_function_` 变量来展示。在这种情况下（当基决策器有 `predict_proba()` 时）决策函数会对每个训练实例返回类别概率。例如，oob 评估预测第二个训练实例有 60.6% 的概率属于正类（39.4% 属于负类）：

```
>>> bag_clf.oob_decision_function_
array([[ 0.,  1.], [ 0.60588235,  0.39411765],[ 1., 0. ],
...  [ 1. ,  0. ],[ 0.,  1.],[ 0.48958333,  0.51041667]])
```

随机贴片与随机子空间

`BaggingClassifier` 也支持采样特征。它被两个超参数 `max_features` 和 `bootstrap_features` 控制。他们的工作方式和 `max_samples` 和 `bootstrap` 一样，但这是对于特征采样而不是实例采样。因此，每一个分类器都会被在随机的输入特征内进行训练。

当你在处理高维度输入下（例如图片）此方法尤其有效。对训练实例和特征的采样被叫做随机贴片。保留了所有的训练实例（例如 `bootstrap=False` 和 `max_samples=1.0`），但是对特征采样（`bootstrap_features=True` 并且/或者 `max_features` 小于 1.0）叫做随机子空间。

采样特征导致更多的预测多样性，用高偏差换低方差。

随机森林

正如我们所讨论的，随机森林是决策树的一种集成，通常是通过 bagging 方法（有时是 pasting 方法）进行训练，通常用 `max_samples` 设置为训练集的大小。与建立一个 `BaggingClassifier` 然后把它放入 *DecisionTreeClassifier* 相反，你可以使用更方便的也是对决策树优化够的 `RandomForestClassifier`（对于回归是 `RandomForestRegressor`）。接下来的代码训练了带有 500 个树（每个被限制为 16 叶子结点）的决策森林，使用所有空闲的 CPU 核：

```
>>>from sklearn.ensemble import RandomForestClassifier
>>>rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16, n_jobs=-1)
>>>rnd_clf.fit(X_train, y_train)
>>>y_pred_rf = rnd_clf.predict(X_test)
```

除了一些例外，`RandomForestClassifier` 使用 `DecisionTreeClassifier` 的所有超参数（决定数怎么生长），把 `BaggingClassifier` 的超参数加起来来控制集成本身。

随机森林算法在树生长时引入了额外的随机；与在节点分裂时需要找到最好分裂特征相反（详见第六章），它在一个随机的特征集中找最好的特征。它导致了树的差异性，并且再一次用高偏差换低方差，总的来说是一个更好的模型。以下是 `BaggingClassifier` 大致相当于之前的 `randomforestclassifier` ：

```
>>>bag_clf = BaggingClassifier(DecisionTreeClassifier(splitter="random", max_leaf_node
s=16),n_estimators=500, max_samples=1.0, bootstrap=True, n_jobs=-1)
```

极端随机树

当你在随机森林上生长树时，在每个结点分裂时只考虑随机特征集上的特征（正如之前讨论过的一样）。相比于找到更好的特征我们可以通过使用对特征使用随机阈值使树更加随机（像规则决策树一样）。

这种极端随机的树被简称为 *Extremely Randomized Trees*（极端随机树），或者更简单的称为 *Extra-Tree*。再一次用高偏差换低方差。它还使得 *Extra-Tree* 比规则的随机森林更快地训练，因为在每个节点上找到每个特征的最佳阈值是生长树最耗时的任务之一。

你可以使用 sklearn 的 `ExtraTreesClassifier` 来创建一个 *Extra-Tree* 分类器。他的 API 跟 `RandomForestClassifier` 是相同的，相似的， *ExtraTreesRegressor* 跟 `RandomForestRegressor` 也是相同的 API。

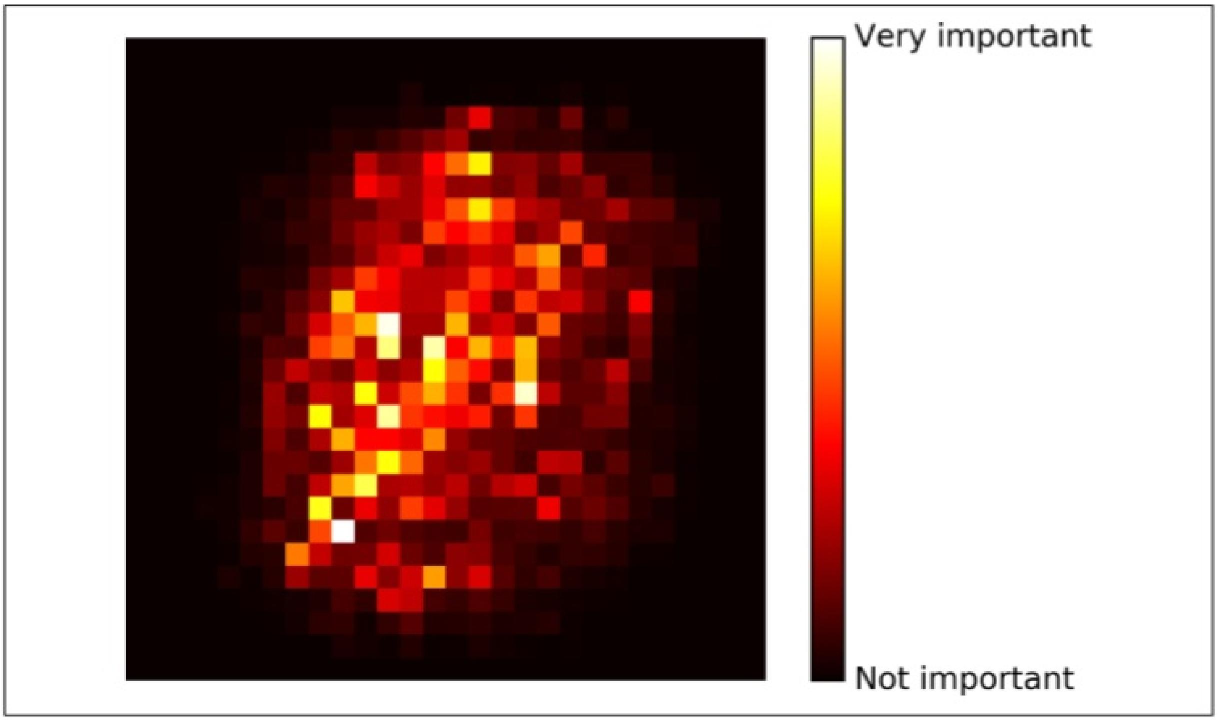
我们很难去分辨 `ExtraTreesClassifier` 和 `RandomForestClassifier` 到底哪个更好。通常情况下是通过交叉验证来比较它们（使用网格搜索调整超参数）。

特征重要度

最后，如果你观察一个单一决策树，重要的特征会出现在更靠近根部的位置，而不重要的特征会经常出现在靠近叶子的位置。因此我们可以通过计算一个特征在森林的全部树中出现的平均深度来预测特征的重要性。sklearn 在训练后会自动计算每个特征的重要度。你可以通过 `feature_importances_` 变量来查看结果。例如如下代码在 iris 数据集（第四章介绍）上训练了一个 `RandomForestClassifier` 模型，然后输出了每个特征的重要性。看来，最重要的特征是花瓣长度（44%）和宽度（42%），而萼片长度和宽度相对比较是不重要的（分别为 11% 和 2%）：

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
>>> rnd_clf.fit(iris["data"], iris["target"])
>>> for name, score in zip(iris["feature_names"], rnd_clf.feature_importances_):
>>>     print(name, score)
sepal length (cm) 0.112492250999
sepal width (cm) 0.0231192882825
petal length (cm) 0.441030464364
petal width (cm) 0.423357996355
```

相似的，如果你在 MNIST 数据及上训练随机森林分类器（在第三章上介绍），然后画出每个像素的重要性，你可以得到图 7-6 的图片。



随机森林可以非常方便快速得了解哪些特征实际上是重要的，特别是你需要进行特征选择的时候。

提升

提升（Boosting，最初称为假设增强）指的是可以将几个弱学习者组合成强学习者的集成方法。对于大多数的提升方法的思想就是按顺序去训练分类器，每一个都要尝试修正前面的分类。现如今已经有很多的提升方法了，但最著名的就是 *Adaboost*（适应性提升，是 *Adaptive Boosting* 的简称） 和 *Gradient Boosting*（梯度提升）。让我们先从 *Adaboost* 说起。

Adaboost

使一个新的分类器去修正之前分类结果的方法就是对之前分类结果不对的训练实例多加关注。这导致新的预测因子越来越多地聚焦于这种情况。这是 *Adaboost* 使用的技术。

举个例子，去构建一个 Adaboost 分类器，第一个基分类器（例如一个决策树）被训练然后在训练集上做预测，在误分类训练实例上的权重就增加了。第二个分类机使用更新过的权重然后再一次训练，权重更新，以此类推（详见图 7-7）

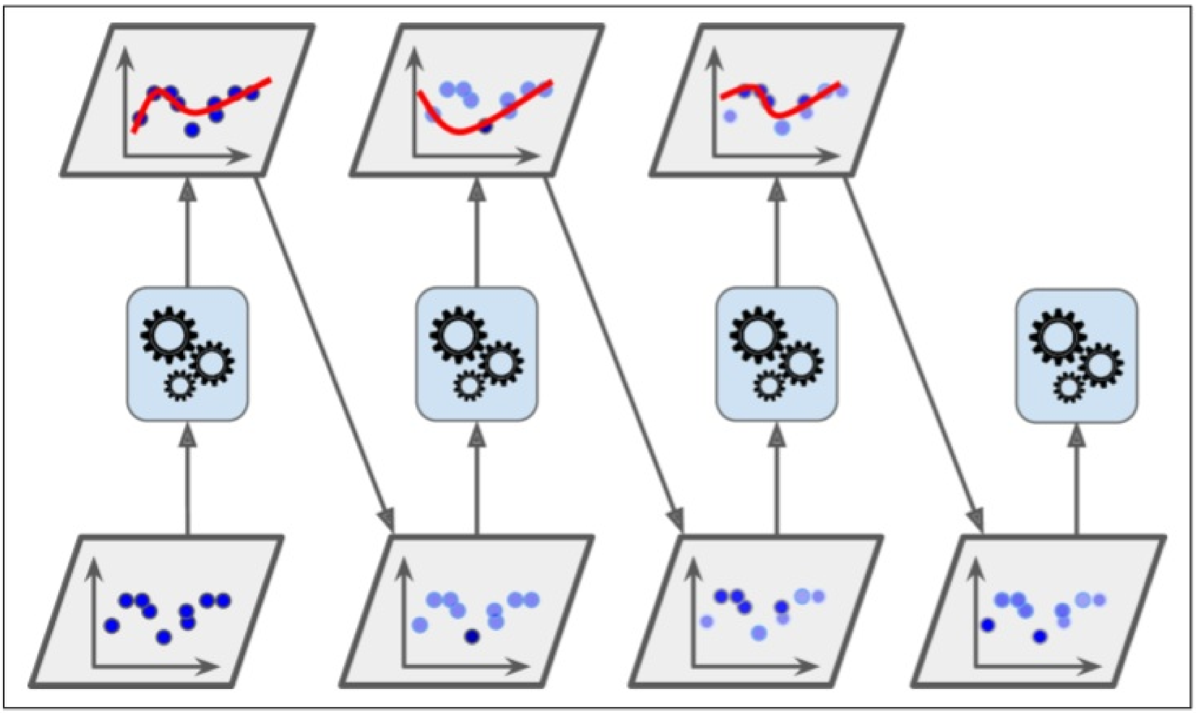
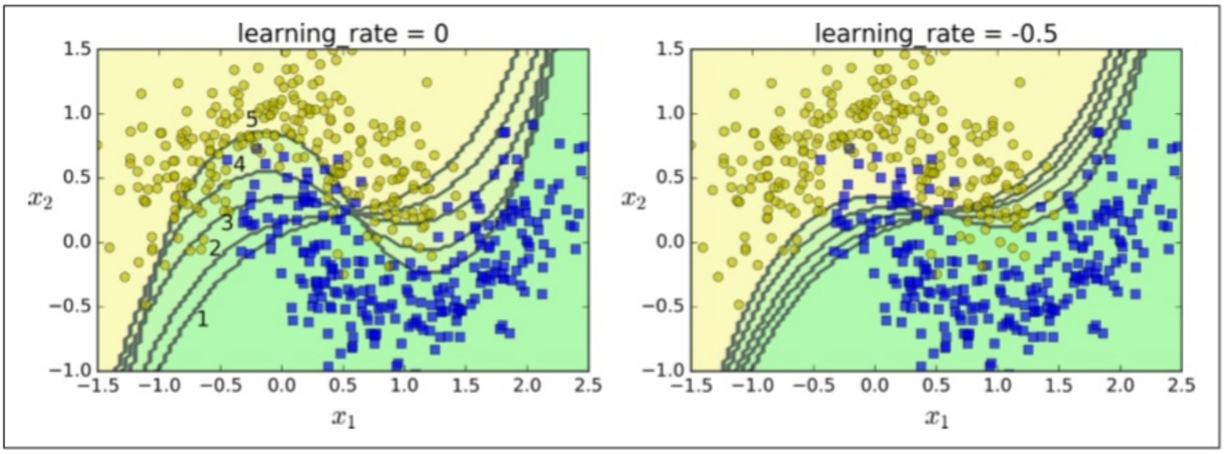


图 7-8 显示连续五次预测的 moons 数据集的决策边界（在本例中，每一个分类器都是高度正则化带有 RBF 核的 SVM）。第一个分类器误分类了很多实例，所以它们的权重被提升了。第二个分类器因此对这些误分类的实例分类效果更好，以此类推。右边的图代表了除了学习率减半外（误分类实例权重每次迭代上升一半）相同的预测序列。你可以看出，序列学习技术与梯度下降很相似，除了调整单个预测因子的参数以最小化代价函数之外，AdaBoost 增加了集合的预测器，逐渐使其更好。



一旦所有的分类器都被训练后，除了分类器根据整个训练集上的准确率被赋予的权重外，集成预测就非常像Bagging和Pasting了。

序列学习技术的一个重要的缺点就是：它不能被并行化（只能按步骤），因为每个分类器只能在之前的分类器已经被训练和评价后再进行训练。因此，它不像Bagging和Pasting一样。

让我们详细看一下 Adaboost 算法。每一个实例的权重`wi`初始都被设为`1/m`第一个分类器被训练，然后他的权重误差率`r1`在训练集上算出，详见公式 7-1。

公式7-1：第`j`个分类器的权重误差率

$$r_j = \frac{\sum_{\substack{i=1 \\ \hat{y}_j^{(i)} \neq y^{(i)}}}^{m} w^{(i)}}{\sum_{i=1}^{m} w^{(i)}}$$

其中 $\widetilde{y_j}^i$ 是第`j`个分类器对于第`i`实例的预测。

分类器的权重 j 随后用公式 7-2 计算出来。其中`η`是超参数学习率（默认为 1）。分类器准确率越高，它的权重就越高。如果它只是瞎猜，那么它的权重会趋近于 0。然而，如果它总是出错（比瞎猜的几率都低），它的权重会使负数。

公式 7-2：分类器权重

$$\alpha_j = \eta \log \frac{1 - r_j}{r_j}$$

接下来实例的权重会按照公式 7-3 更新：误分类的实例权重会被提升。

公式7-3 权重更新规则

对于`i=1, 2, ..., m`

$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{if } \widehat{y_j}^{(i)} = y^{(i)} \\ w^{(i)} \exp\left(\alpha_j\right) & \text{if } \widehat{y_j}^{(i)} \neq y^{(i)} \end{cases}$$

随后所有实例的权重都被归一化（例如被 $\sum_{i=1}^{m} w^i$ 整除）

最后，一个新的分类器通过更新过的权重训练，整个过程被重复（新的分类器权重被计算，实例的权重被更新，随后另一个分类器被训练，以此类推）。当规定的分类器数量达到或者最好的分类器被找到后算法就会停止。

为了进行预测，Adaboost 通过分类器权重 j 简单的计算了所有的分类器和权重。预测类别会是权重投票中主要的类别。（详见公式 7-4）

公式7-4： Adaboost 分类器

$$\hat{y}(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \sum_{\substack{j=1 \\ \hat{y}_j(\mathbf{x}) = k}}^{N} \alpha_j$$

其中 `N` 是分类器的数量。

sklearn 通常使用 Adaboost 的多分类版本 *SAMME*（这就代表了 分段加建模使用多类指数损失函数）。如果只有两类别，那么 *SAMME* 是与 Adaboost 相同的。如果分类器可以预测类别概率（例如如果它们有 `predict_proba()` ），如果 sklearn 可以使用 *SAMME* 叫做 `SAMME.R` 的变量（R 代表“REAL”），这种依赖于类别概率的通常比依赖于分类器的更好。

接下来的代码训练了使用 sklearn 的 `AdaBoostClassifier` 基于 200 个决策树桩 Adaboost 分类器（正如你说期待的，对于回归也有 `AdaBoostRegressor` ）。一个决策树桩是 `max_depth=1` 的决策树-换句话说，是一个单一的决策节点加上两个叶子结点。这就是 `AdaBoostClassifier` 的默认基分类器：

```
>>>from sklearn.ensemble import AdaBoostClassifier
>>>ada_clf = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1), n_estimators=200,
algorithm="SAMME.R", learning_rate=0.5)
>>>ada_clf.fit(X_train, y_train)
```

如果你的 Adaboost 集成过拟合了训练集，你可以尝试减少基分类器的数量或者对基分类器使用更强的正则化。

梯度提升

另一个非常著名的提升算法是梯度提升。与 Adaboost 一样，梯度提升也是通过向集成中逐步增加分类器运行的，每一个分类器都修正之前的分类结果。然而，它并不像 Adaboost 那样每一次迭代都更改实例的权重，这个方法是去使用新的分类器去拟合前面分类器预测的残差 。

让我们通过一个使用决策树当做基分类器的简单的回归例子（回归当然也可以使用梯度提升）。这被叫做梯度提升回归树（GBRT，*Gradient Tree Boosting* 或者 *Gradient Boosted Regression Trees*）。首先我们用 `DecisionTreeRegressor` 去拟合训练集（例如一个有噪二次训练集）：

```
>>>from sklearn.tree import DecisionTreeRegressor
>>>tree_reg1 = DecisionTreeRegressor(max_depth=2)
>>>tree_reg1.fit(X, y)
```

现在在第一个分类器的残差上训练第二个分类器：

```
>>>y2 = y - tree_reg1.predict(X)
>>>tree_reg2 = DecisionTreeRegressor(max_depth=2)
>>>tree_reg2.fit(X, y2)
```

随后在第二个分类器的残差上训练第三个分类器：

```
>>>y3 = y2 - tree_reg1.predict(X)
>>>tree_reg3 = DecisionTreeRegressor(max_depth=2)
>>>tree_reg3.fit(X, y3)
```

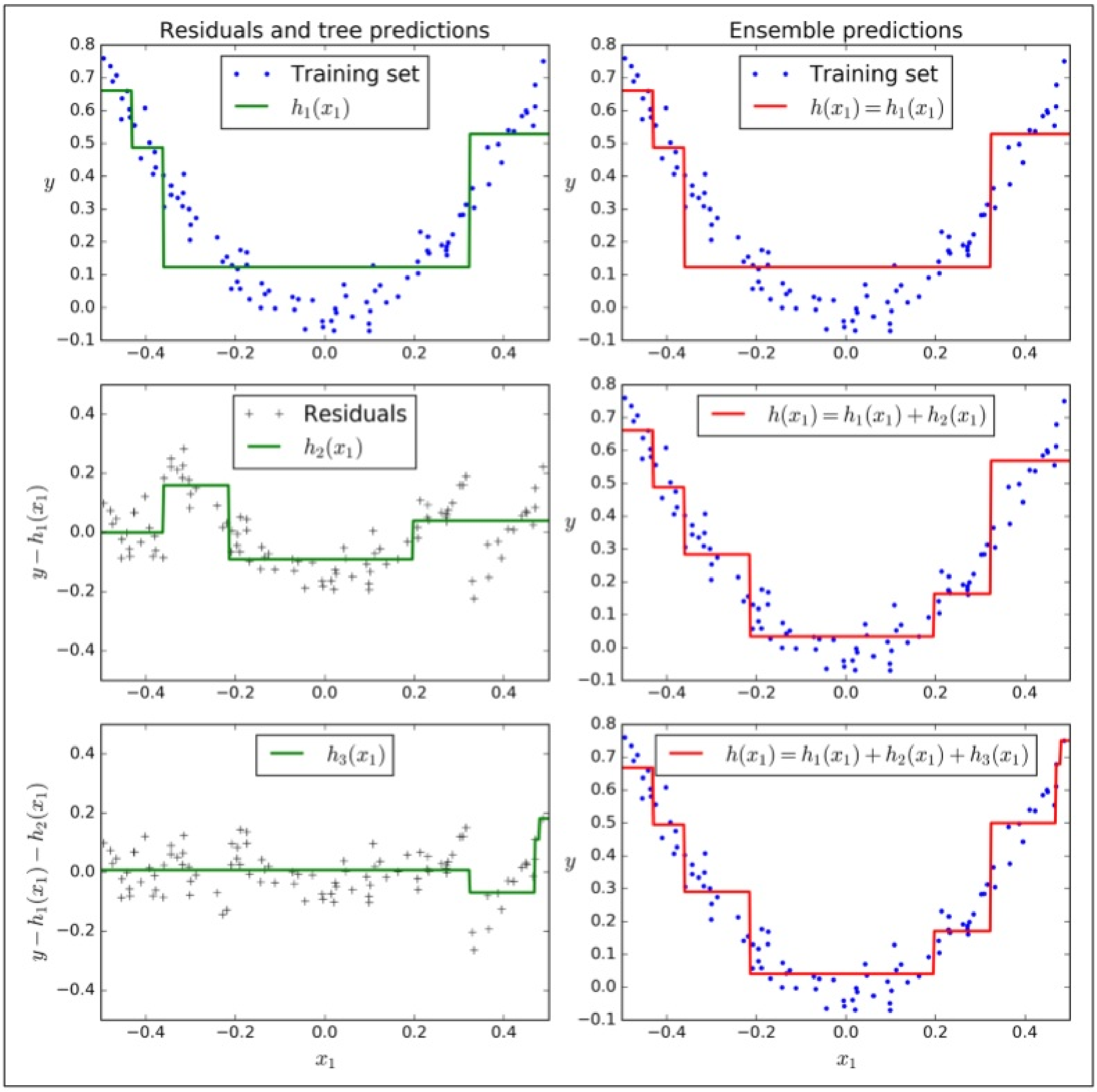
现在我们有了一个包含三个回归器的集成。它可以通过集成所有树的预测来在一个新的实例上进行预测。

```
>>>y_pred = sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
```

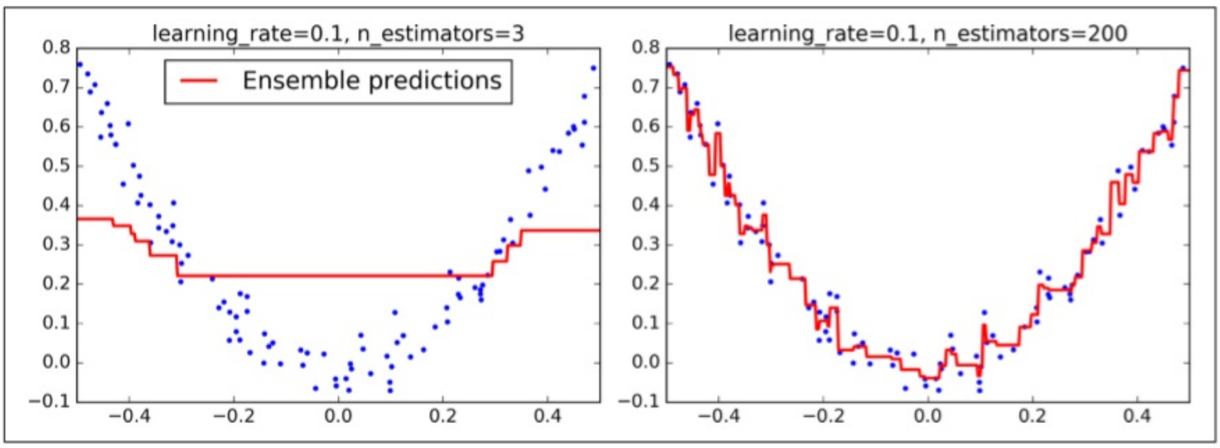
图7-9在左栏展示了这三个树的预测，在右栏展示了集成的预测。在第一行，集成只有一个树，所以它与第一个树的预测相似。在第二行，一个新的树在第一个树的残差上进行训练。在右边栏可以看出集成的预测等于前两个树预测的和。相同的，在第三行另一个树在第二个数的残差上训练。你可以看到集成的预测会变的更好。

我们可以使用 sklean 中的 `GradientBoostingRegressor` 来训练 GBRT 集成。与 `RandomForestClassifier` 相似，它也有超参数去控制决策树的生长（例如 `max_depth`，`min_samples_leaf` 等等），也有超参数去控制集成训练，例如基分类器的数量（`n_estimators`）。接下来的代码创建了与之前相同的集成：

```
>>>from sklearn.ensemble import GradientBoostingRegressor
>>>gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3, learning_rate=1.0)
>>>gbrt.fit(X, y)
```



超参数 `learning_rate` 确立了每个树的贡献。如果你把它设置为一个很小的树，例如 0.1，在集成中就需要更多的树去拟合训练集，但预测通常会更好。这个正则化技术叫做 *shrinkage*。图 7-10 展示了两个在低学习率上训练的 GBRT 集成：其中左面是一个没有足够树去拟合训练集的树，右面是有过多的树过拟合训练集的树。

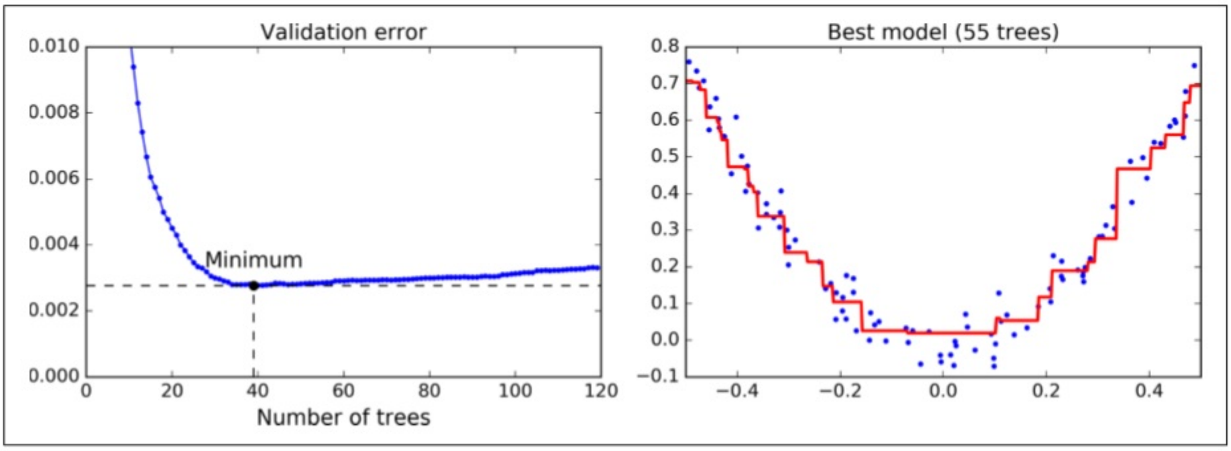


为了找到树的最优数量，你可以使用早停技术（第四章讨论）。最简单使用这个技术的方法就是使用 `staged_predict()`：它在训练的每个阶段（用一棵树，两棵树等）返回一个迭代器。加下来的代码用 120 个树训练了一个 GBRT 集成，然后在训练的每个阶段验证错误以找到树的最佳数量，最后使用 GBRT 树的最优数量训练另一个集成：

```
>>>import numpy as np
>>>from sklearn.model_selection import train_test_split
>>>from sklearn.metrics import mean_squared_error

>>>X_train, X_val, y_train, y_val = train_test_split(X, y)
>>>gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=120)
>>>gbrt.fit(X_train, y_train)
>>>errors = [mean_squared_error(y_val, y_pred)
     for y_pred in gbrt.staged_predict(X_val)]
>>>bst_n_estimators = np.argmin(errors)
>>>gbrt_best = GradientBoostingRegressor(max_depth=2,n_estimators=bst_n_estimators)
>>>gbrt_best.fit(X_train, y_train)
```

验证错误在图 7-11 的左面展示，最优模型预测被展示在右面。



你也可以早早的停止训练来实现早停（与先在一大堆树中训练，然后再回头去找最优数目相反）。你可以通过设置 `warm_start=True` 来实现 ，这使得当 `fit()` 方法被调用时 sklearn 保留现有树，并允许增量训练。接下来的代码在当一行中的五次迭代验证错误没有改善时会停止训练：

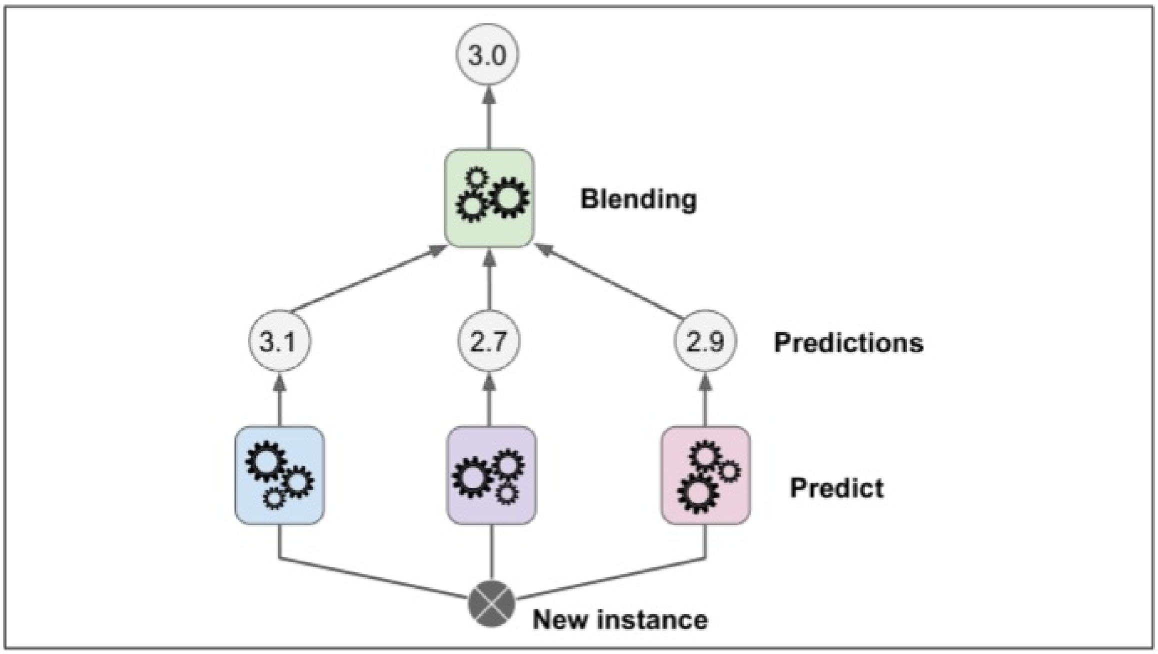
```
>>>gbrt = GradientBoostingRegressor(max_depth=2, warm_start=True)
min_val_error = float("inf")
error_going_up = 0
for n_estimators in range(1, 120):
    gbrt.n_estimators = n_estimators
    gbrt.fit(X_train, y_train)
    y_pred = gbrt.predict(X_val)
    val_error = mean_squared_error(y_val, y_pred)
    if val_error < min_val_error:
        min_val_error = val_error
        error_going_up = 0
    else:
        error_going_up += 1
        if error_going_up == 5:
            break  # early stopping
```

`GradientBoostingRegressor` 也支持指定用于训练每棵树的训练实例比例的超参数 `subsample` 。例如如果 `subsample=0.25` ，那么每个树都会在 25% 随机选择的训练实例上训练。你现在也能猜出来，这也是个高偏差换低方差的作用。它同样也加速了训练。这个技术叫做随机梯度提升。

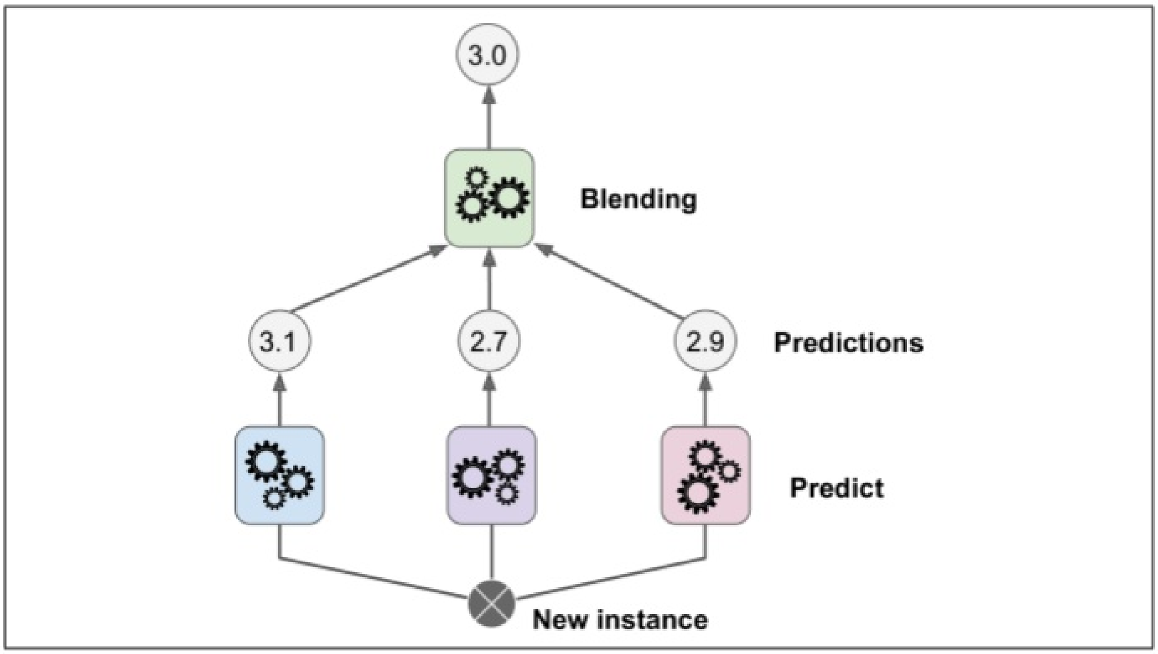
也可能对其他损失函数使用梯度提升。这是由损失超参数控制（见 sklearn 文档）。

Stacking

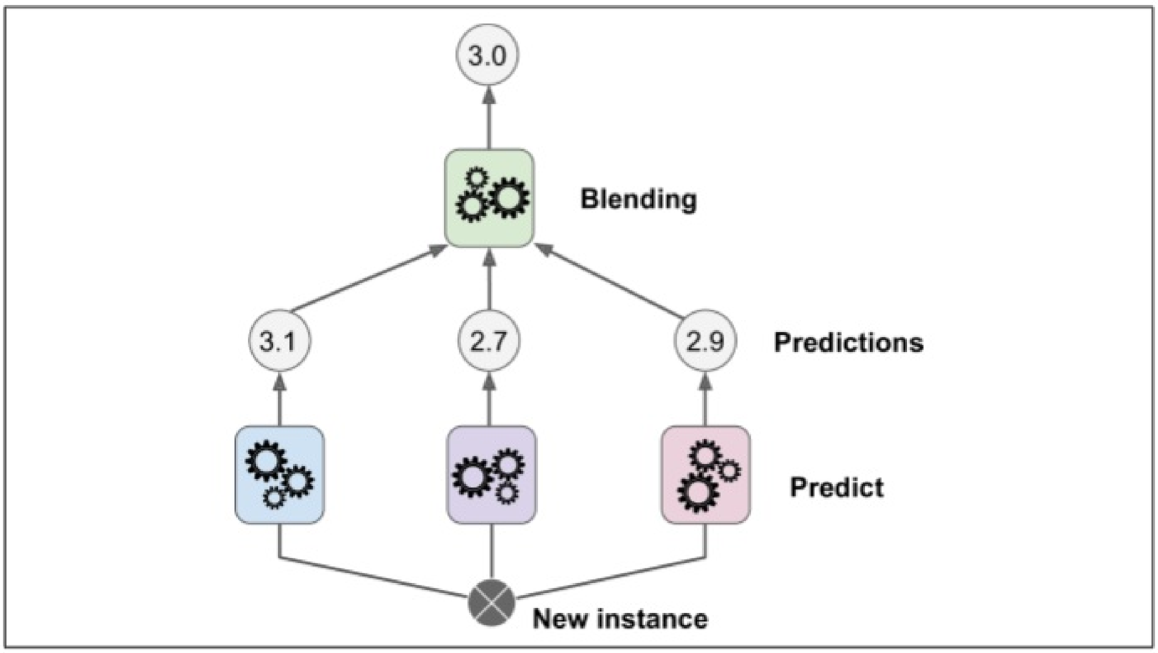
本章讨论的最后一个集成方法叫做 *Stacking*（*stacked generalization* 的缩写）。这个算法基于一个简单的想法：不使用琐碎的函数（如硬投票）来聚合集合中所有分类器的预测，我们为什么不训练一个模型来执行这个聚合？图 7-12 展示了这样一个在新的回归实例上预测的集成。底部三个分类器每一个都有不同的值（3.1，2.7 和 2.9），然后最后一个分类器（叫做 *blender* 或者 *meta learner* ）把这三个分类器的结果当做输入然后做出最终决策（3.0）。



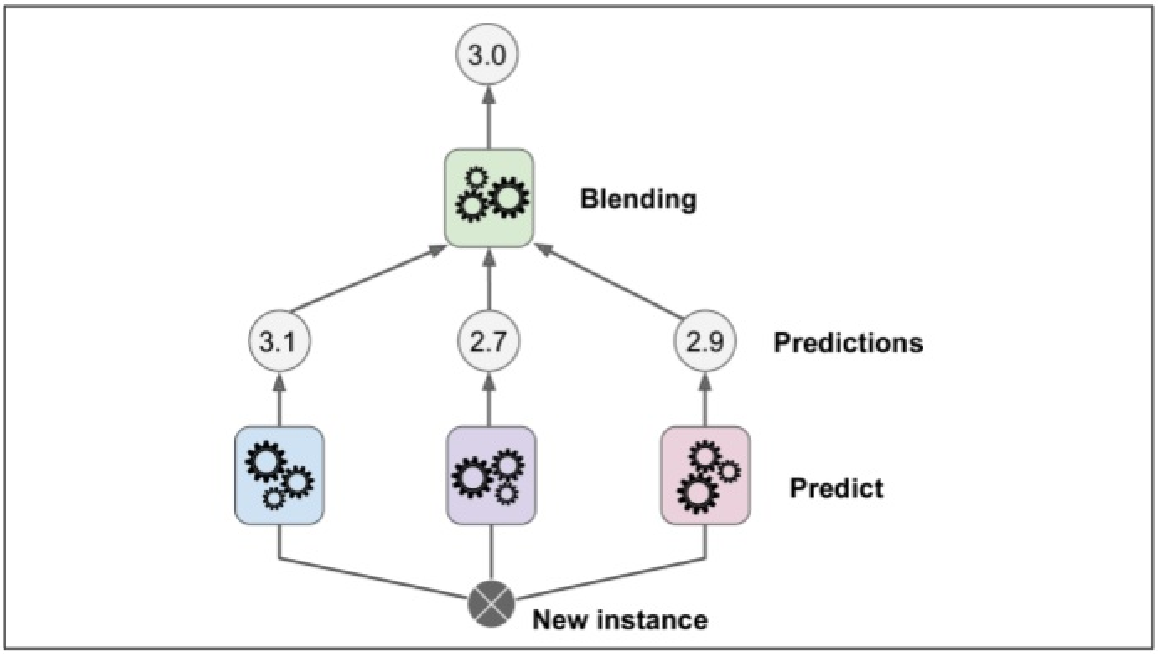
为了训练这个 *blender* ，一个通用的方法是采用保持集。让我们看看它怎么工作。首先，训练集被分为两个子集，第一个子集被用作训练第一层（详见图 7-13）.



接下来，第一层的分类器被用来预测第二个子集（保持集）（详见 7-14）。这确保了预测结果很“干净”，因为这些分类器在训练的时候没有使用过这些事例。现在对在保持集中的每一个实例都有三个预测值。我们现在可以使用这些预测结果作为输入特征来创建一个新的训练集（这使得这个训练集是三维的），并且保持目标数值不变。随后 *blender* 在这个新的训练集上训练，因此，它学会了预测第一层预测的目标值。



显然我们可以用这种方法训练不同的 *blender* （例如一个线性回归，另一个是随机森林等等）：我们得到了一层 *blender* 。诀窍是将训练集分成三个子集：第一个子集用来训练第一层，第二个子集用来创建训练第二层的训练集（使用第一层分类器的预测值），第三个子集被用来创建训练第三层的训练集（使用第二层分类器的预测值）。以上步骤做完了，我们可以通过逐个遍历每个层来预测一个新的实例。详见图 7-15.



然而不幸的是，sklearn 并不直接支持 stacking ，但是你自己组建是很容易的（看接下来的练习）。或者你也可以使用开源的项目例如 *brew* （网址为 https://github.com/viisar/brew）

练习

1. 如果你在相同训练集上训练 5 个不同的模型，它们都有 95% 的准确率，那么你是否可以通过组合这个模型来得到更好的结果？如果可以那怎么做呢？如果不可以请给出理由。
2. 软投票和硬投票分类器之间有什么区别？
3. 是否有可能通过分配多个服务器来加速 bagging 集成系统的训练？pasting 集成，boosting 集成，随机森林，或 stacking 集成怎么样？
4. out-of-bag 评价的好处是什么？
5. 是什么使 Extra-Tree 比规则随机森林更随机呢？这个额外的随机有什么帮助呢？那这个 Extra-Tree 比规则随机森林谁更快呢？
6. 如果你的 Adaboost 模型欠拟合，那么你需要怎么调整超参数？
7. 如果你的梯度提升过拟合，那么你应该调高还是调低学习率呢？
8. 导入 MNIST 数据（第三章中介绍），把它切分进一个训练集，一个验证集，和一个测试集（例如 40000 个实例进行训练，10000 个进行验证，10000 个进行测试）。然后训练多个分类器，例如一个随机森林分类器，一个 Extra-Tree 分类器和一个 SVM。接下来，尝试将它们组合成集成，使用软或硬投票分类器来胜过验证集上的所有集合。一旦找到了，就在测试集上实验。与单个分类器相比，它的性能有多好？
9. 从练习 8 中运行个体分类器来对验证集进行预测，并创建一个新的训练集并生成预测：每个训练实例是一个向量，包含来自所有分类器的图像的预测集，目标是图像类别。祝贺你，你刚刚训练了一个 *blender* ，和分类器一起组成了一个叠加组合！现在让我们来评估测试集上的集合。对于测试集中的每个图像，用所有分类器进行预测，然后将预测馈送到 *blender* 以获得集合的预测。它与你早期训练过的投票分类器相比如何？

练习的答案都在附录A上。

第8章 降维

很多机器学习的问题都会涉及到有着几千甚至数百万维的特征的训练实例。这不仅让训练过程变得非常缓慢，同时还很难找到一个很好的解，我们接下来就会遇到这种情况。这种问题通常被称为维数灾难（curse of dimentionality）。

幸运的是，在现实生活中我们经常可以极大的降低特征维度，将一个十分棘手的问题转变成一个可以较为容易解决的问题。例如，对于 MNIST 图片集（第 3 章中提到）：图片四周边缘部分的像素几乎总是白的，因此你完全可以将这些像素从你的训练集中扔掉而不会丢失太多信息。图 7-6 向我们证实了这些像素的确对我们的分类任务是完全不重要的。同时，两个相邻的像素往往是高度相关的：如果你想要将他们合并成一个像素（比如取这两个像素点的平均值）你并不会丢失很多信息。

> 警告：降维肯定会丢失一些信息（这就好比将一个图片压缩成 JPEG 的格式会降低图像的质量），因此即使这种方法可以加快训练的速度，同时也会让你的系统表现的稍微差一点。降维会让你的工作流水线更复杂因而更难维护。所有你应该先尝试使用原始的数据来训练，如果训练速度太慢的话再考虑使用降维。在某些情况下，降低训练集数据的维度可能会筛选掉一些噪音和不必要的细节，这可能会让你的结果比降维之前更好（这种情况通常不会发生；它只会加快你训练的速度）。

降维除了可以加快训练速度外，在数据可视化方面（或者 DataViz）也十分有用。降低特征维度到 2（或者 3）维从而可以在图中画出一个高维度的训练集，让我们可以通过视觉直观的发现一些非常重要的信息，比如聚类。

在这一章里，我们将会讨论维数灾难问题并且了解在高维空间的数据。然后，我们将会展示两种主要的降维方法：投影（projection）和流形学习（Manifold Learning），同时我们还会介绍三种流行的降维技术：主成分分析（PCA），核主成分分析（Kernel PCA）和局部线性嵌入（LLE）。

维数灾难

我们已经习惯生活在一个三维的世界里，以至于当我们尝试想象更高维的空间时，我们的直觉不管用了。即使是一个基本的 4D 超正方体也很难在我们的脑中想象出来（见图 8-1），更不用说一个 200 维的椭球弯曲在一个 1000 维的空间里了。

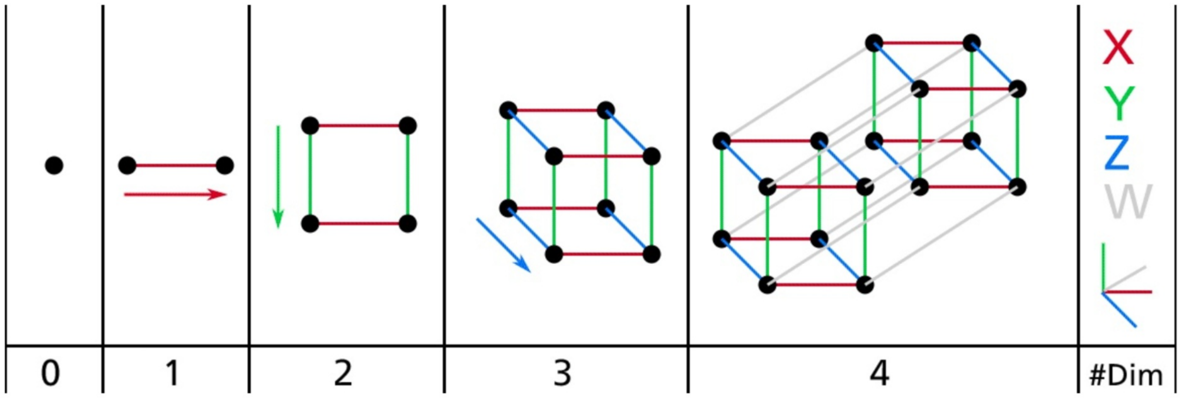


图 8-1 点，线，方形，立方体和超正方体（0D 到 4D 超正方体）

这表明很多物体在高维空间表现的十分不同。比如，如果你在一个正方形单元中随机取一个点（一个 `1×1` 的正方形），那么随机选的点离所有边界大于 0.001（靠近中间位置）的概率为 0.4%（ `1 - 0.998^2` ）（换句话说，一个随机产生的点不大可能严格落在某一个维度上。但是在一个 1,0000 维的单位超正方体（一个 `1×1×...×1` 的立方体，有 10,000 个 1），这种可能性超过了 99.999999%。在高维超正方体中，大多数点都分布在边界处。

还有一个更麻烦的区别：如果你在一个平方单位中随机选取两个点，那么这两个点之间的距离平均约为 0.52。如果您在单位 3D 立方体中选取两个随机点，平均距离将大致为 0.66。但是，在一个 1,000,000 维超立方体中随机抽取两点呢？那么，平均距离，信不信由你，大概为 408.25（大致 $\sqrt{1,000,000/6}$）！这非常违反直觉：当它们都位于同一单元超立方体内时，两点是怎么距离这么远的？这一事实意味着高维数据集有很大风险分布的非常稀疏：大多数训练实例可能彼此远离。当然，这也意味着一个新实例可能远离任何训练实例，这使得预测的可靠性远低于我们处理较低维度数据的预测，因为它们将基于更大的推测（extrapolations）。简而言之，训练集的维度越高，过拟合的风险就越大。

理论上来说，维数爆炸的一个解决方案是增加训练集的大小从而达到拥有足够密度的训练集。不幸的是，在实践中，达到给定密度所需的训练实例的数量随着维度的数量呈指数增长。如果只有 100 个特征（比 MNIST 问题要少得多）并且假设它们均匀分布在所有维度上，那么如果想要各个临近的训练实例之间的距离在 0.1 以内，您需要比宇宙中的原子还要多的训练实例。

降维的主要方法

在我们深入研究具体的降维算法之前，我们来看看降低维度的两种主要方法：投影和流形学习。

投影（Projection）

在大多数现实生活的问题中，训练实例并不是在所有维度上均匀分布的。许多特征几乎是常数，而其他特征则高度相关（如前面讨论的 MNIST）。结果，所有训练实例实际上位于（或接近）高维空间的低维子空间内。这听起来有些抽象，所以我们不妨来看一个例子。在图 8-2 中，您可以看到由圆圈表示的 3D 数据集。

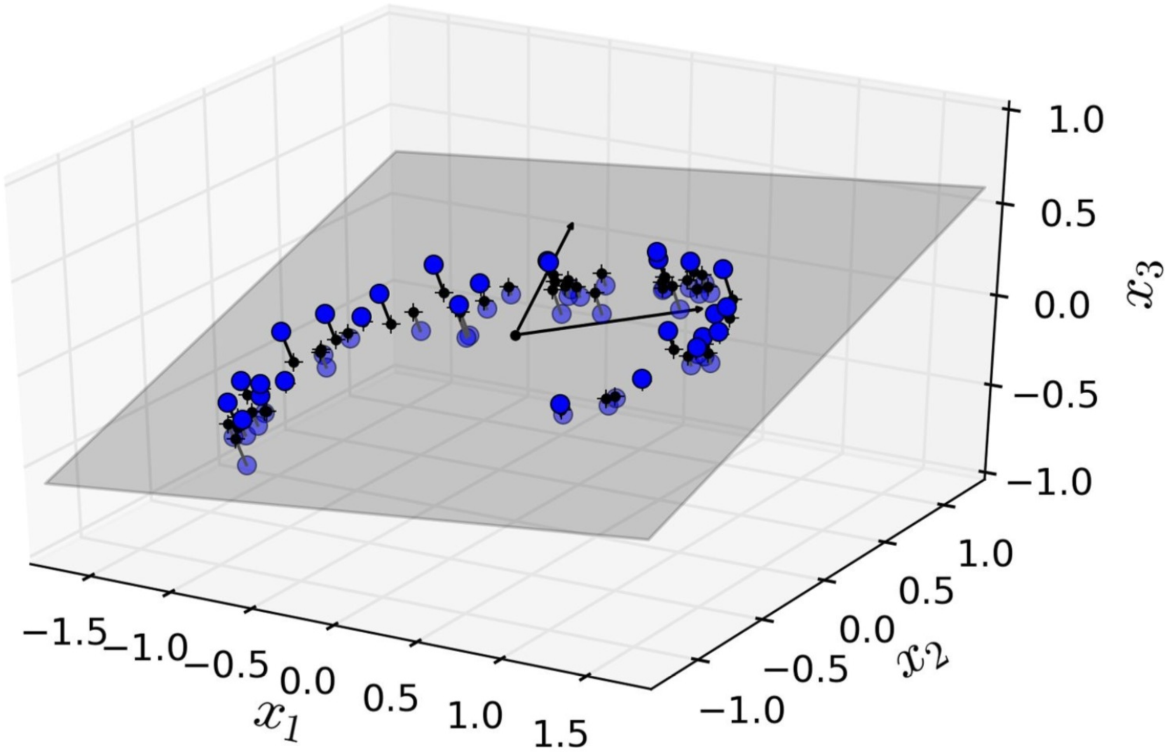


图 8-2 一个分布接近于2D子空间的3D数据集

注意到所有训练实例的分布都贴近一个平面：这是高维（3D）空间的较低维（2D）子空间。现在，如果我们将每个训练实例垂直投影到这个子空间上（就像将短线连接到平面的点所表示的那样），我们就可以得到如图8-3所示的新2D数据集。铛铛铛！我们刚刚将数据集的维度从 3D 降低到了 2D。请注意，坐标轴对应于新的特征 `z1` 和 `z2` （平面上投影的坐标）。

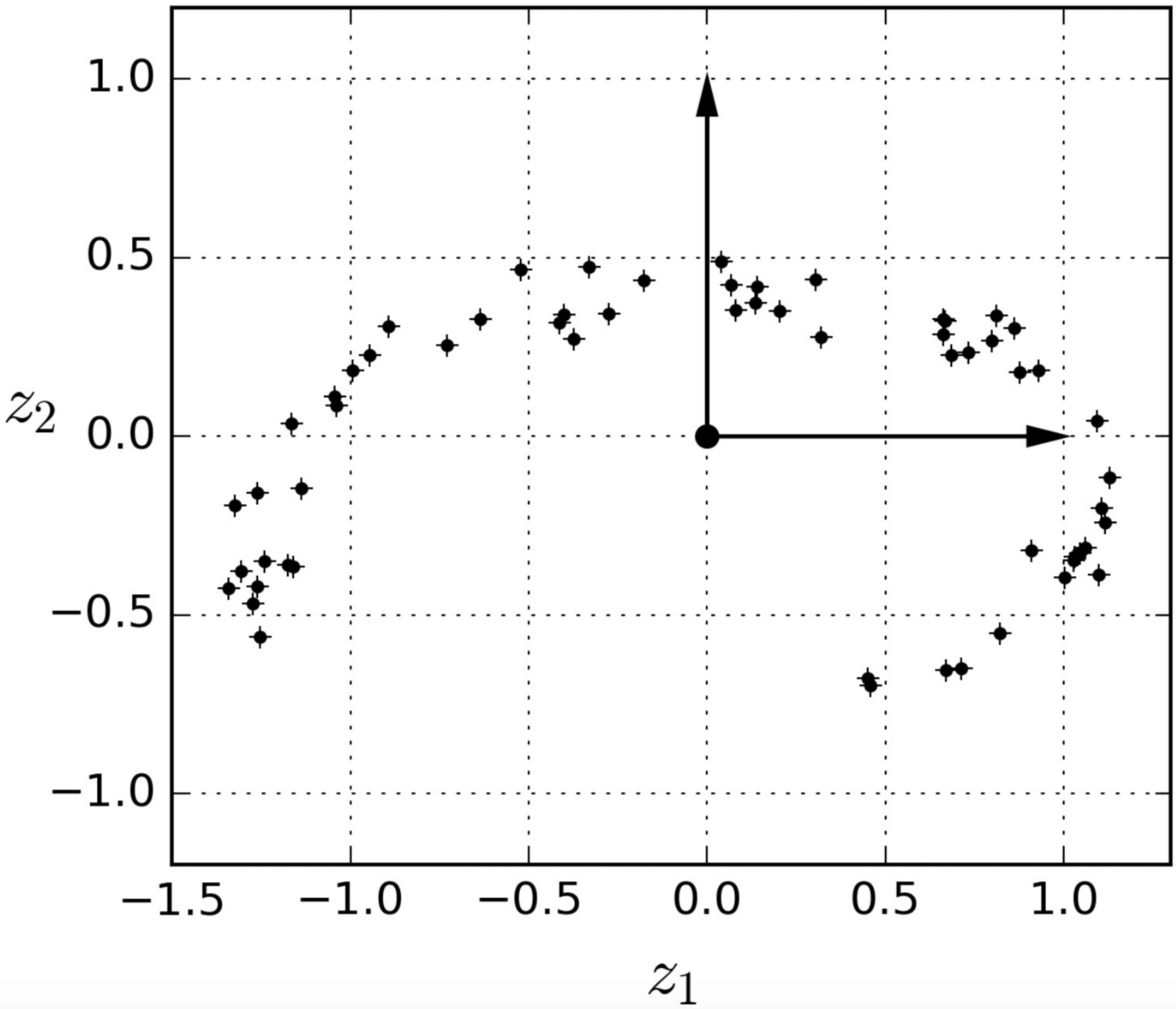


图 8-3 一个经过投影后的新的 2D 数据集

但是，投影并不总是降维的最佳方法。在很多情况下，子空间可能会扭曲和转动，比如图 8-4 所示的着名瑞士滚动玩具数据集。

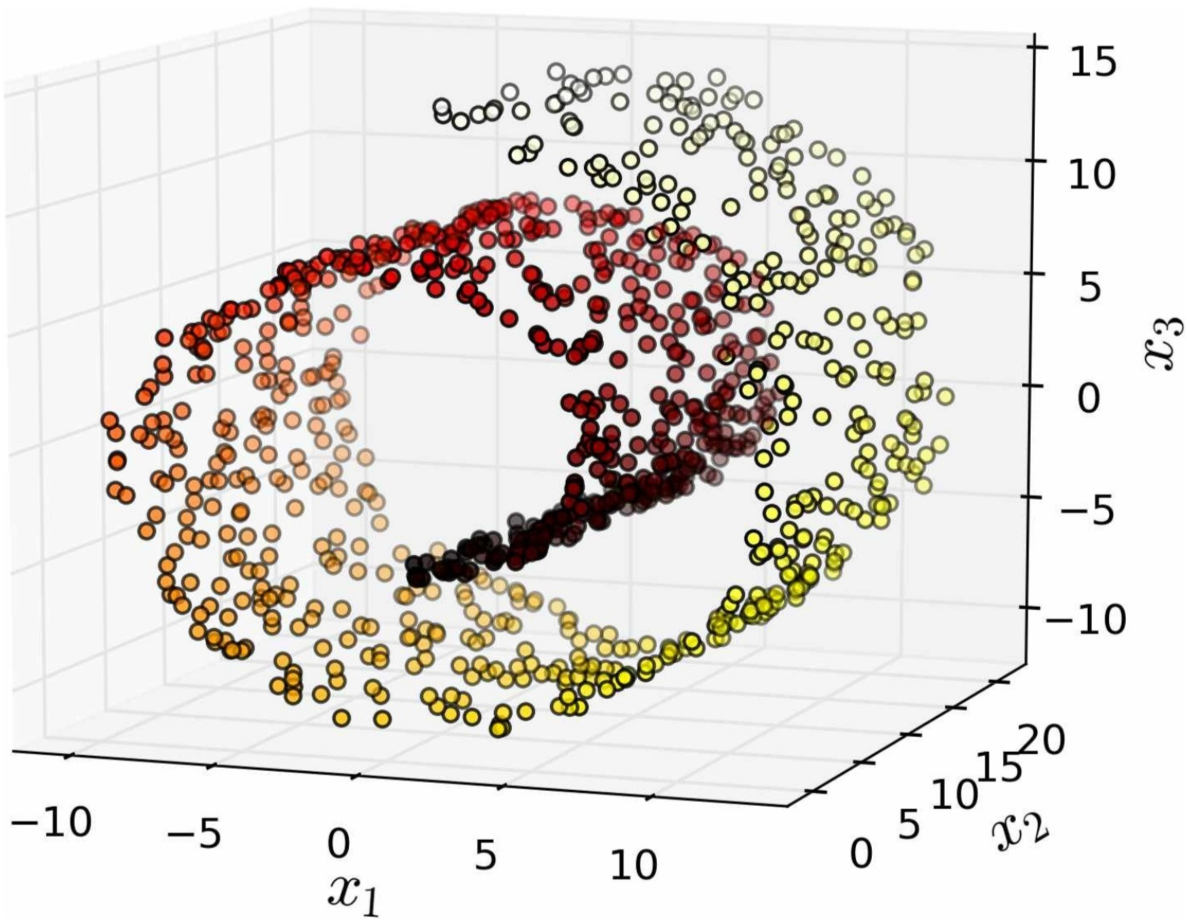


图 8-4 瑞士滚动数玩具数据集

简单地将数据集投射到一个平面上（例如，直接丢弃 `x3` ）会将瑞士卷的不同层叠在一起，如图 8-5 左侧所示。但是，你真正想要的是展开瑞士卷所获取到的类似图 8-5 右侧的 2D 数据集。

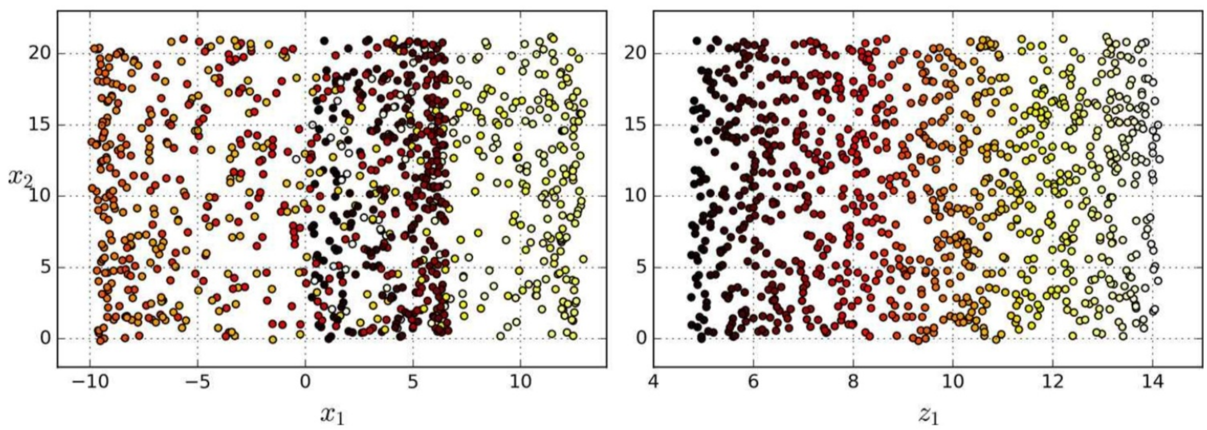


图 8-5 投射到平面的压缩（左）vs 展开瑞士卷（右）

流形学习

瑞士卷一个是二维流形的例子。简而言之，二维流形是一种二维形状，它可以在更高维空间中弯曲或扭曲。更一般地，一个 `d` 维流形是类似于 `d` 维超平面的 `n` 维空间（其中 `d < n`）的一部分。在我们瑞士卷这个例子中，`d = 2`，`n = 3`：它有些像 2D 平面，但是它实际上是在第三维中卷曲。

许多降维算法通过对训练实例所在的流形进行建模从而达到降维目的；这叫做流形学习。它依赖于流形猜想（manifold assumption），也被称为流形假设（manifold hypothesis），它认为大多数现实世界的高维数据集大都靠近一个更低维的流形。这种假设经常在实践中被证实。

让我们再回到 MNIST 数据集：所有手写数字图像都有一些相似之处。它们由连线组成，边界是白色的，大多是在图片中中间的，等等。如果你随机生成图像，只有一小部分看起来像手写数字。换句话说，如果您尝试创建数字图像，那么您的自由度远低于您生成任何随便一个图像时的自由度。这些约束往往会将数据集压缩到较低维流形中。

流形假设通常包含着另一个隐含的假设：你现在的手上的工作（例如分类或回归）如果在流形的较低维空间中表示，那么它们会变得更简单。例如，在图 8-6 的第一行中，瑞士卷被分为两类：在三维空间中（图左上），分类边界会相当复杂，但在二维展开的流形空间中（图右上），分类边界是一条简单的直线。

但是，这个假设并不总是成立。例如，在图 8-6 的最下面一行，决策边界位于 `x1 = 5`（图左下）。这个决策边界在原始三维空间（一个垂直平面）看起来非常简单，但在展开的流形中却变得更复杂了（四个独立线段的集合）（图右下）。

简而言之，如果在训练模型之前降低训练集的维数，那训练速度肯定会加快，但并不总是会得出更好的训练效果；这一切都取决于数据集。

希望你现在对于维数爆炸以及降维算法如何解决这个问题有了一定的理解，特别是对流形假设提出的内容。本章的其余部分将介绍一些最流行的降维算法。

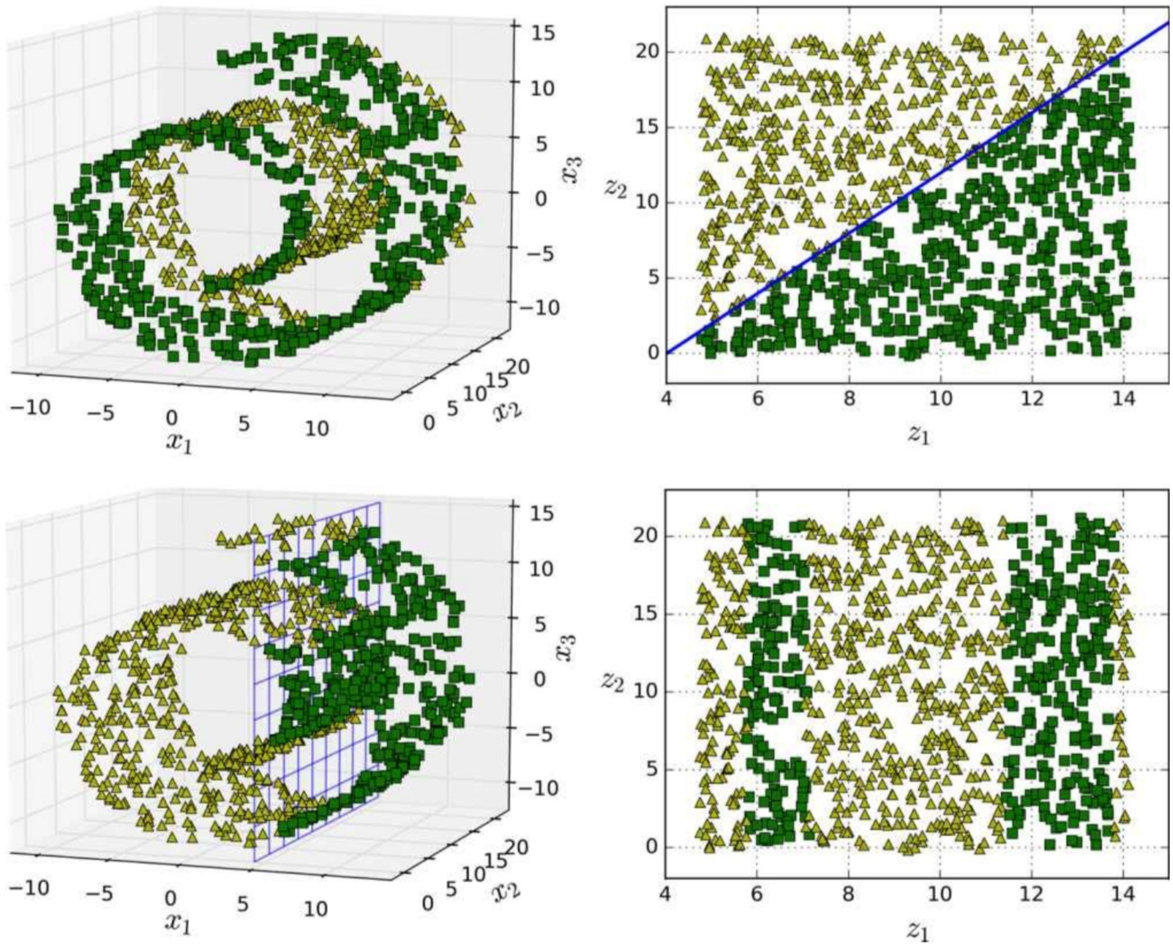


图 8-6 决策边界并不总是会在低维空间中变的简单

主成分分析（PCA）

主成分分析（Principal Component Analysis）是目前为止最流行的降维算法。首先它找到接近数据集分布的超平面，然后将所有的数据都投影到这个超平面上。

保留（最大）方差

在将训练集投影到较低维超平面之前，您首先需要选择正确的超平面。例如图 8-7 左侧是一个简单的二维数据集，以及三个不同的轴（即一维超平面）。图右边是将数据集投影到每个轴上的结果。正如你所看到的，投影到实线上保留了最大方差，而在点线上的投影只保留了非常小的方差，投影到虚线上保留的方差则处于上述两者之间。

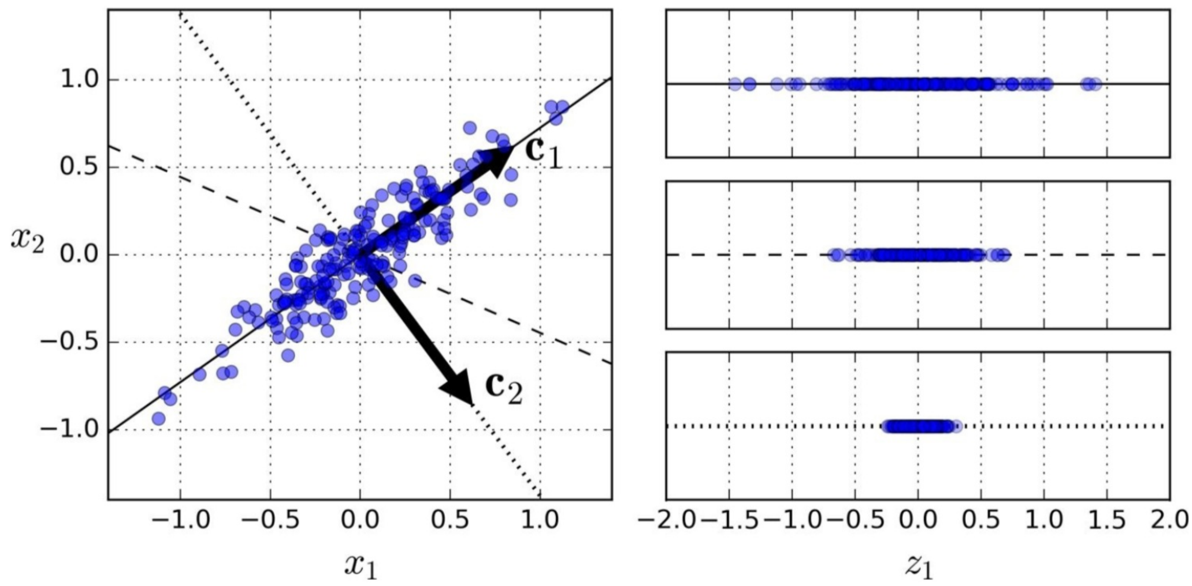


图 8-7 选择投射到哪一个子空间

选择保持最大方差的轴看起来是合理的，因为它很可能比其他投影损失更少的信息。证明这种选择的另一种方法是，选择这个轴使得将原始数据集投影到该轴上的均方距离最小。这是就 PCA 背后的思想，相当简单。

主成分（Principle Componets）

PCA 寻找训练集中可获得最大方差的轴。在图 8-7 中，它是一条实线。它还发现了一个与第一个轴正交的第二个轴，选择它可以获得最大的残差。在这个 2D 例子中，没有选择：就只有这条点线。但如果在一个更高维的数据集中，PCA 也可以找到与前两个轴正交的第三个轴，以及与数据集中维数相同的第四个轴，第五个轴等。 定义第 `i` 个轴的单位矢量被称为第 `i` 个主成分（PC）。在图 8-7 中，第一个 PC 是 `c1`，第二个 PC 是 `c2`。在图 8-2 中，前两个 PC 用平面中的正交箭头表示，第三个 PC 与上述 PC 形成的平面正交（指向上或下）。

> 概述： 主成分的方向不稳定：如果您稍微打乱一下训练集并再次运行 PCA，则某些新 PC 可能会指向与原始 PC 方向相反。但是，它们通常仍位于同一轴线上。在某些情况下，一对 PC 甚至可能会旋转或交换，但它们定义的平面通常保持不变。

那么如何找到训练集的主成分呢？幸运的是，有一种称为奇异值分解（SVD）的标准矩阵分解技术，可以将训练集矩阵 `X` 分解为三个矩阵 `U·Σ·V^T` 的点积，其中 `V^T` 包含我们想要的所有主成分，如公式 8-1 所示。

公式 8-1 主成分矩阵

$$\mathbf{V}^T = \begin{pmatrix} | & | & & | \\ \mathbf{c_1} & \mathbf{c_2} & \cdots & \mathbf{c_n} \\ | & | & & | \end{pmatrix}$$

下面的 Python 代码使用了 Numpy 提供的 `svd()` 函数获得训练集的所有主成分，然后提取前两个 PC:

```
X_centered=X-X.mean(axis=0)
U,s,V=np.linalg.svd(X_centered)
c1=V.T[:,0]
c2=V.T[:,1]
```

> 警告：PCA 假定数据集以原点为中心。正如我们将看到的，Scikit-Learn 的 `PCA` 类负责为您的数据集中心化处理。但是，如果您自己实现 PCA（如前面的示例所示），或者如果您使用其他库，不要忘记首先要先对数据做中心化处理。

投影到 `d` 维空间

一旦确定了所有的主成分，你就可以通过将数据集投影到由前 `d` 个主成分构成的超平面上，从而将数据集的维数降至 `d` 维。选择这个超平面可以确保投影将保留尽可能多的方差。例如，在图 8-2 中，3D 数据集被投影到由前两个主成分定义的 2D 平面，保留了大部分数据集的方差。因此，2D 投影看起来非常像原始 3D 数据集。

为了将训练集投影到超平面上，可以简单地通过计算训练集矩阵 `X` 和 `Wd` 的点积，`Wd` 定义为包含前 `d` 个主成分的矩阵（即由 `V^T` 的前 `d` 列组成的矩阵），如公式 8-2 所示。

公式 8-2 将训练集投影到 `d` 维空间

$$X_{d-proj} = X \cdot W_d$$

下面的 Python 代码将训练集投影到由前两个主成分定义的超平面上：

```
W2=V.T[:,:2]
X2D=X_centered.dot(W2)
```

好了你已经知道这个东西了！你现在已经知道如何给任何一个数据集降维而又能尽可能的保留原数据集的方差了。

使用 Scikit-Learn

Scikit-Learn 的 PCA 类使用 SVD 分解来实现，就像我们之前做的那样。以下代码应用 PCA 将数据集的维度降至两维（请注意，它会自动处理数据的中心化）：

```
from sklearn.decomposition import PCA

pca=PCA(n_components=2)
X2D=pca.fit_transform(X)
```

将 PCA 转化器应用于数据集后，可以使用 `components_` 访问每一个主成分（注意，它返回以 PC 作为水平向量的矩阵，因此，如果我们想要获得第一个主成分则可以写成 `pca.components_.T[:,0]` ）。

方差解释率（Explained Variance Ratio）

另一个非常有用的信息是每个主成分的方差解释率，可通过 `explained_variance_ratio_` 变量获得。它表示位于每个主成分轴上的数据集方差的比例。例如，让我们看一下图 8-2 中表示的三维数据集前两个分量的方差解释率：

```
>>> print(pca.explained_variance_ratio_)
array([0.84248607, 0.14631839])
```

这表明，84.2% 的数据集方差位于第一轴，14.6% 的方差位于第二轴。第三轴的这一比例不到1.2％，因此可以认为它可能没有包含什么信息。

选择正确的维度

通常我们倾向于选择加起来到方差解释率能够达到足够占比（例如 95%）的维度的数量，而不是任意选择要降低到的维度数量。当然，除非您正在为数据可视化而降低维度 -- 在这种情况下，您通常希望将维度降低到 2 或 3。

下面的代码在不降维的情况下进行 PCA，然后计算出保留训练集方差 95% 所需的最小维数：

```
pca=PCA()
pac.fit(X)
cumsum=np.cumsum(pca.explained_variance_ratio_)
d=np.argmax(cumsum>=0.95)+1
```

你可以设置 `n_components = d` 并再次运行 PCA。但是，有一个更好的选择：不指定你想要保留的主成分个数，而是将 `n_components` 设置为 0.0 到 1.0 之间的浮点数，表明您希望保留的方差比率：

```
pca=PCA(n_components=0.95)
X_reduced=pca.fit_transform(X)
```

另一种选择是画出方差解释率关于维数的函数（简单地绘制 `cumsum` ；参见图 8-8）。曲线中通常会有一个肘部，方差解释率停止快速增长。您可以将其视为数据集的真正的维度。在这种情况下，您可以看到将维度降低到大约100个维度不会失去太多的可解释方差。

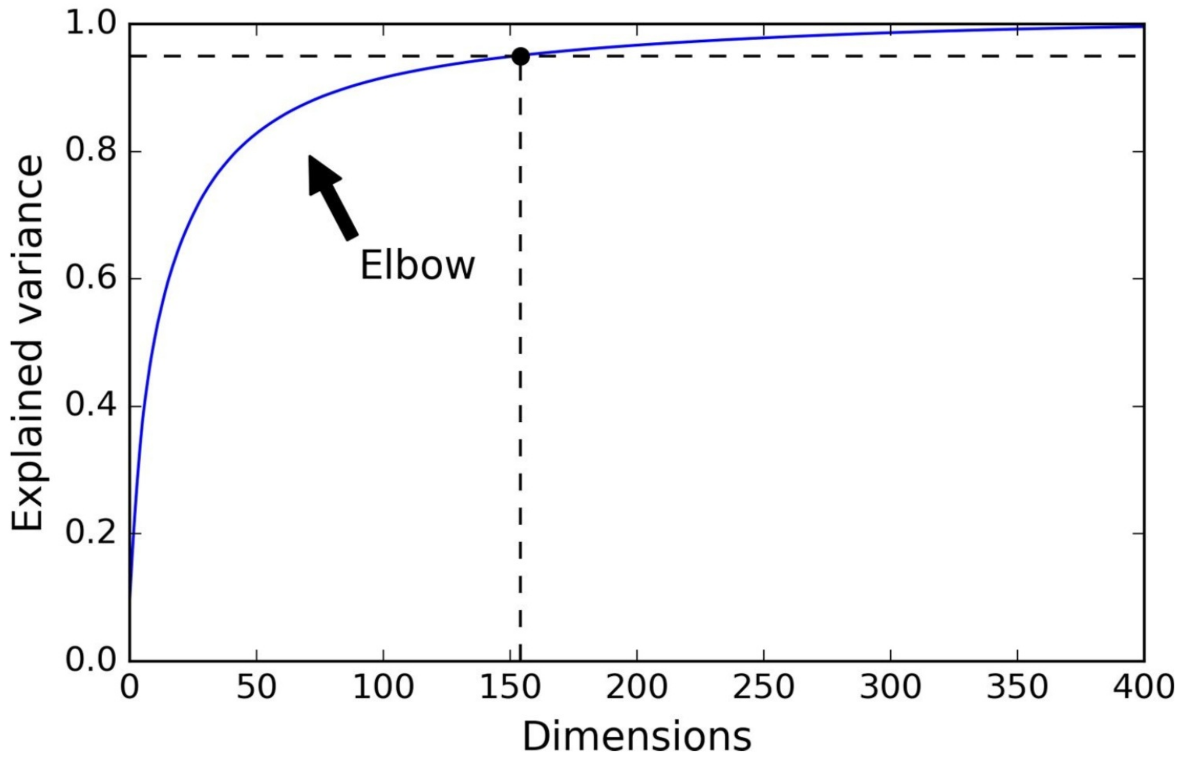


图 8-8 可解释方差关于维数的函数

PCA 压缩

显然，在降维之后，训练集占用的空间要少得多。例如，尝试将 PCA 应用于 MNIST 数据集，同时保留 95% 的方差。你应该发现每个实例只有 150 多个特征，而不是原来的 784 个特征。因此，尽管大部分方差都保留下来，但数据集现在还不到其原始大小的 20%！这是一个合理的压缩比率，您可以看到这可以如何极大地加快分类算法（如 SVM 分类器）的速度。

通过应用 PCA 投影的逆变换，也可以将缩小的数据集解压缩回 784 维。当然这并不会返回给你最原始的数据，因为投影丢失了一些信息（在5%的方差内），但它可能非常接近原始数据。原始数据和重构数据之间的均方距离（压缩然后解压缩）被称为重构误差（reconstruction error）。例如，下面的代码将 MNIST 数据集压缩到 154 维，然后使用 `inverse_transform()` 方法将其解压缩回 784 维。图 8-9 显示了原始训练集（左侧）的几位数字在压缩并解压缩后（右侧）的对应数字。您可以看到有轻微的图像质量降低，但数字仍然大部分完好无损。

```
pca=PCA(n_components=154)
X_mnist_reduced=pca.fit_transform(X_mnist)
X_mnist_recovered=pca.inverse_transform(X_mnist_reduced)
```

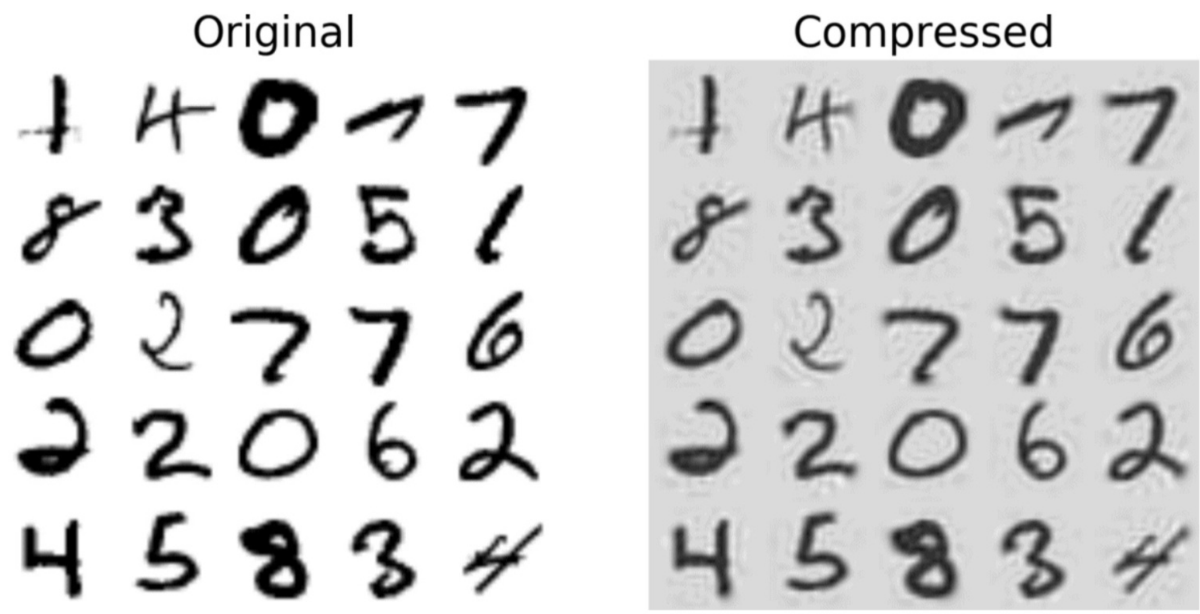


图 8-9 MNIST 保留 95 方差的压缩

逆变换的公式如公式 8-3 所示

公式 8-3 PCA逆变换，回退到原来的数据维度

$$X_{recovered} = X_{d-proj} \cdot W_d^T$$

增量 PCA（Incremental PCA）

先前 PCA 实现的一个问题是它需要在内存中处理整个训练集以便 SVD 算法运行。幸运的是，我们已经开发了增量 PCA（IPCA）算法：您可以将训练集分批，并一次只对一个批量使用 IPCA 算法。这对大型训练集非常有用，并且可以在线应用 PCA（即在新实例到达时即时运行）。

下面的代码将 MNIST 数据集分成 100 个小批量（使用 NumPy 的 `array_split()` 函数），并将它们提供给 Scikit-Learn 的 `IncrementalPCA` 类，以将 MNIST 数据集的维度降低到 154 维（就像以前一样）。请注意，您必须对每个最小批次调用 `partial_fit()` 方法，而不是对整个训练集使用 `fit()` 方法：

```
from sklearn.decomposition import IncrementalPCA

n_batches=100
inc_pca=IncrementalPCA(n_components=154)
for X_batch in np.array_spplit(X_mnist,n_batches):
    inc_pca.partial_fit(X_batch)
X_mnist_reduced=inc_pca.transform(X_mnist)
```

或者，您可以使用 NumPy 的 `memmap` 类，它允许您操作存储在磁盘上二进制文件中的大型数组，就好像它完全在内存中；该类仅在需要时加载内存中所需的数据。由于增量 PCA 类在任何时间内仅使用数组的一小部分，因此内存使用量仍受到控制。这可以调用通常的 `fit()` 方

法，如下面的代码所示：

```
X_mm=np.memmap(filename,dtype='float32',mode='readonly',shape=(m,n))
batch_size=m//n_batches
inc_pca=IncrementalPCA(n_components=154,batch_size=batch_size)
inc_pca.fit(X_mm)
```

随机 PCA（Randomized PCA）

Scikit-Learn 提供了另一种执行 PCA 的选择，称为随机 PCA。这是一种随机算法，可以快速找到前d个主成分的近似值。它的计算复杂度是 `O(m × d^2) + O(d^3)` ，而不是 `O(m × n^2) + O(n^3)` ，所以当 `d` 远小于 `n` 时，它比之前的算法快得多。

```
rnd_pca=PCA(n_components=154,svd_solver='randomized')
X_reduced=rnd_pca.fit_transform(X_mnist)
```

核 PCA（Kernel PCA）

在第 5 章中，我们讨论了核技巧，一种将实例隐式映射到非常高维空间（称为特征空间）的数学技术，让支持向量机可以应用于非线性分类和回归。回想一下，高维特征空间中的线性决策边界对应于原始空间中的复杂非线性决策边界。

事实证明，同样的技巧可以应用于 PCA，从而可以执行复杂的非线性投影来降低维度。这就是所谓的核 PCA（kPCA）。它通常能够很好地保留投影后的簇，有时甚至可以展开分布近似于扭曲流形的数据集。

例如，下面的代码使用 Scikit-Learn 的 `KernelPCA` 类来执行带有 RBF 核的 kPCA（有关 RBF 核和其他核的更多详细信息，请参阅第 5 章）：

```
from sklearn.decomposition import KernelPCA

rbf_pca=KernelPCA(n_components=2,kernel='rbf',gamma=0.04)
X_reduced=rbf_pca.fit_transform(X)
```

图 8-10 展示了使用线性核（等同于简单的使用 PCA 类），RBF 核，sigmoid 核（Logistic）将瑞士卷降到 2 维。

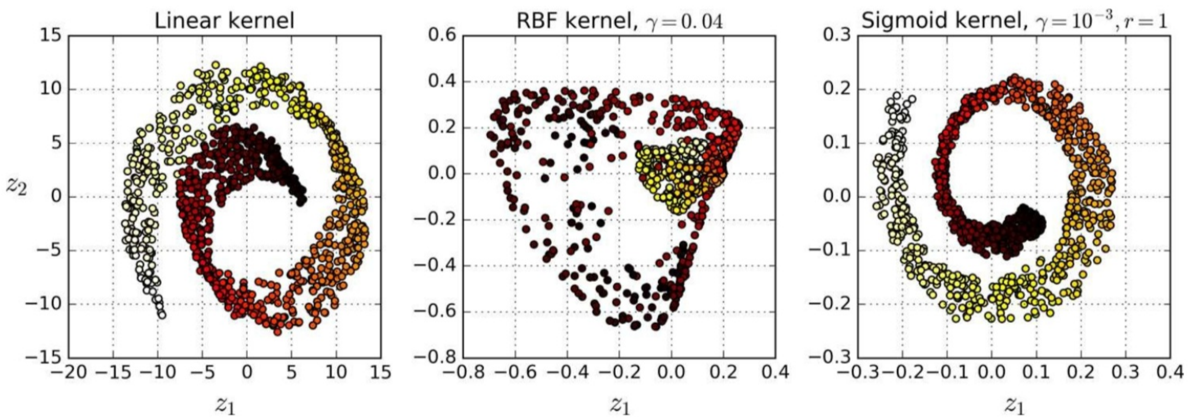


图 8-10 使用不同核的 kPCA 将瑞士卷降到 2 维

选择一种核并调整超参数

由于 kPCA 是无监督学习算法，因此没有明显的性能指标可以帮助您选择最佳的核方法和超参数值。但是，降维通常是监督学习任务（例如分类）的准备步骤，因此您可以简单地使用网格搜索来选择可以让该任务达到最佳表现的核方法和超参数。例如，下面的代码创建了一个两步的流水线，首先使用 kPCA 将维度降至两维，然后应用 Logistic 回归进行分类。然后它使用 `Grid SearchCV` 为 kPCA 找到最佳的核和 `gamma` 值，以便在最后获得最佳的分类准确性：

```
from sklearn.model_selection import GridSearchCV from sklearn.linear_model import Logi
sticRegression from sklearn.pipeline import Pipeline
clf = Pipeline([
        ("kpca", KernelPCA(n_components=2)),
        ("log_reg", LogisticRegression())
])
param_grid = [{
        "kpca__gamma": np.linspace(0.03, 0.05, 10),
        "kpca__kernel": ["rbf", "sigmoid"]
    }]
grid_search = GridSearchCV(clf, param_grid, cv=3)
grid_search.fit(X, y)
```

你可以通过调用 `best_params_` 变量来查看使模型效果最好的核和超参数：

```
>>> print(grid_search.best_params_)
{'kpca__gamma': 0.043333333333333335, 'kpca__kernel': 'rbf'}
```

另一种完全为非监督的方法，是选择产生最低重建误差的核和超参数。但是，重建并不像线性 PCA 那样容易。这里是原因：图 8-11 显示了原始瑞士卷 3D 数据集（左上角），并且使用 RBF 核应用 kPCA 后生成的二维数据集（右上角）。由于核技巧，这在数学上等同于使用特征映射 `φ` 将训练集映射到无限维特征空间（右下），然后使用线性 PCA 将变换的训练集投影到 2D。请注意，如果我们可以在缩减空间中对给定实例实现反向线性 PCA 步骤，则重构点将位于特征空间中，而不是位于原始空间中（例如，如图中由 `x` 表示的那样）。由于特征空间是无限维的，我们不能找出重建点，因此我们无法计算真实的重建误差。幸运的是，可以

在原始空间中找到一个贴近重建点的点。这被称为重建前图像（reconstruction pre-image）。一旦你有这个前图像，你就可以测量其与原始实例的平方距离。然后，您可以选择最小化重建前图像错误的核和超参数。

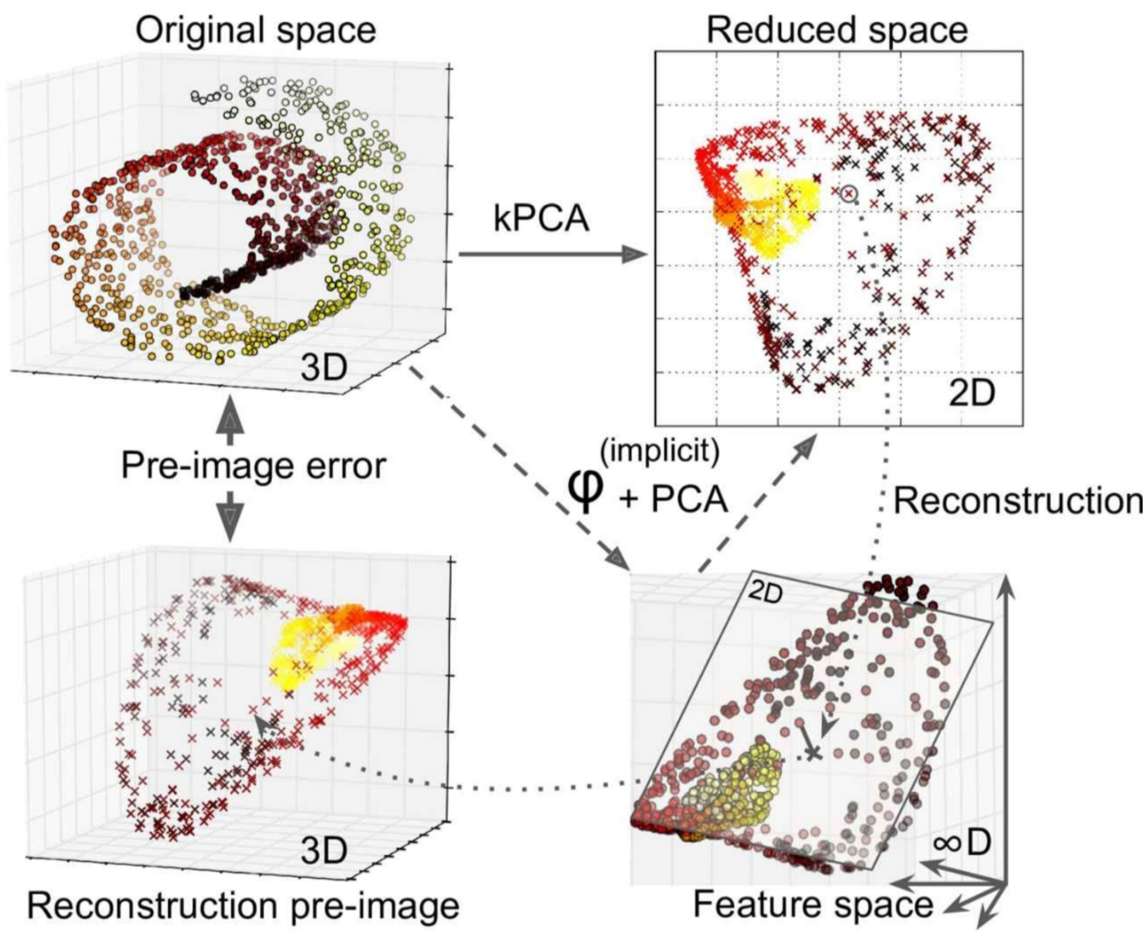


图 8-11 核 PCA 和重建前图像误差

您可能想知道如何进行这种重建。一种解决方案是训练一个监督回归模型，将预计实例作为训练集，并将原始实例作为训练目标。如果您设置了 `fit_inverse_transform = True` ，Scikit-Learn 将自动执行此操作，代码如下所示：

```
rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.0433,fit_inverse_transform=
True)
X_reduced = rbf_pca.fit_transform(X)
X_preimage = rbf_pca.inverse_transform(X_reduced)
```

> 概述：默认条件下， `fit_inverse_transform = False` 并且 `KernelPCA` 没有 `inverse_tranfrom()` 方法。这种方法仅仅当 `fit_inverse_transform = True` 的情况下才会创建。

你可以计算重建前图像误差：

```
>>> from sklearn.metrics import mean_squared_error
>>> mean_squared_error(X, X_preimage) 32.786308795766132
```

现在你可以使用交叉验证的方格搜索来寻找可以最小化重建前图像误差的核方法和超参数。

LLE

局部线性嵌入（Locally Linear Embedding）是另一种非常有效的非线性降维（NLDR）方法。这是一种流形学习技术，不依赖于像以前算法那样的投影。简而言之，LLE 首先测量每个训练实例与其最近邻（c.n.）之间的线性关系，然后寻找能最好地保留这些局部关系的训练集的低维表示（稍后会详细介绍） 。这使得它特别擅长展开扭曲的流形，尤其是在没有太多噪音的情况下。

例如，以下代码使用 Scikit-Learn 的 `LocallyLinearEmbedding` 类来展开瑞士卷。得到的二维数据集如图 8-12 所示。正如您所看到的，瑞士卷被完全展开，实例之间的距离保存得很好。但是，距离不能在较大范围内保留的很好：展开的瑞士卷的左侧被挤压，而右侧的部分被拉长。尽管如此，LLE 在对流形建模方面做得非常好。

```
from sklearn.manifold import LocallyLinearEmbedding

lle=LocallyLinearEmbedding(n_components=2,n_neighbors=10)
X_reduced=lle.fit_transform(X)
```

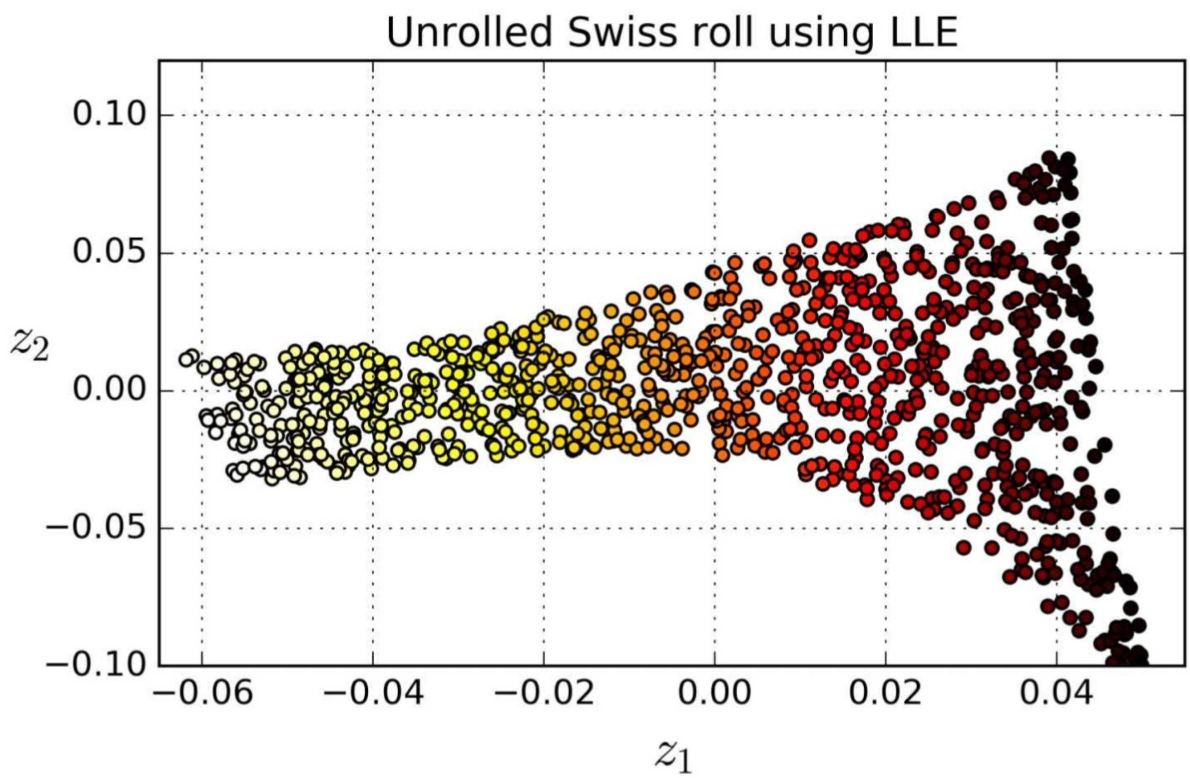


图 8-12 使用 LLE 展开瑞士卷

这是LLE的工作原理：首先，对于每个训练实例 $x^{(i)}$，该算法识别其最近的 `k` 个邻居（在前面的代码中 `k = 10` 中），然后尝试将 $x^{(i)}$ 重构为这些邻居的线性函数。更具体地，找到权重 $w_{i,j}$ 从而使 $x^{(i)}$ 和 $\sum_{j=1}^{m} w_{i,j}x^{(j)}$ 之间的平方距离尽可能的小，假设如果 $x^{(j)}$ 不是 $x^{(i)}$ 的 `k` 个最近邻时 $w_{i,j} = 0$。因此，LLE 的第一步是方程 8-4 中描述的约束优化问题，其中 `W` 是包含所有权重 $w_{i,j}$ 的权重矩阵。第二个约束简单地对每个训练实例 $x^{(i)}$ 的权重进行归一化。

公式 8-4 LLE 第一步：对局部关系进行线性建模

$$
\widehat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmin}} \sum_{i=1}^{m} \left\| \mathbf{x}^{(i)} - \sum_{j=1}^{m} w_{i,j}\mathbf{x}^{(j)} \right\|^2
$$

$$
\text{subject to } \begin{cases} w_{i,j} = 0 & \text{if } \mathbf{x}^{(j)} \text{ is not one of the } k \text{ c.n. of } \mathbf{x}^{(i)} \\ \sum_{j=1}^{m} w_{i,j} = 1 & \text{for } i = 1, 2, \cdots, m \end{cases}
$$

在这步之后，权重矩阵 $\widehat{W}$（包含权重 $\hat{w_{i,j}}$ 对训练实例的线形关系进行编码。现在第二步是将训练实例投影到一个 `d` 维空间（ `d < n` ）中去，同时尽可能的保留这些局部关系。如果 $z^{(i)}$ 是 $x^{(i)}$ 在这个 `d` 维空间的图像，那么我们想要 $z^{(i)}$ 和 $\sum_{j=1}^{m} \hat{w_{i,j}}\ z^{(j)}$ 之间的平方距离尽可能的小。这个想法让我们提出了公式8-5中的非限制性优化问题。它看起来与第一步非常相似，但我们要做的不是保持实例固定并找到最佳权重，而是恰相反：保持权重不变，并在低维空间中找到实例图像的最佳位置。请注意， `Z` 是包含所有 $z^{(i)}$ 的矩阵。

公式 8-5 LLE 第二步：保持关系的同时进行降维

$$
\widehat{\mathbf{Z}} = \underset{\mathbf{Z}}{\operatorname{argmin}} \sum_{i=1}^{m} \left\| \mathbf{z}^{(i)} - \sum_{j=1}^{m} \widehat{w}_{i,j}\mathbf{z}^{(j)} \right\|^2
$$

Scikit-Learn 的 LLE 实现具有如下的计算复杂度：查找 `k` 个最近邻为 `O(m log(m) n log(k))` ，优化权重为 `O(m n k^3)` ，建立低维表示为 `O(d m^2)` 。不幸的是，最后一项 `m^2` 使得这个算法在处理大数据集的时候表现较差。

其他降维方法

还有很多其他的降维方法，Scikit-Learn 支持其中的好几种。这里是其中最流行的：

- 多维缩放（MDS）在尝试保持实例之间距离的同时降低了维度（参见图 8-13）
- Isomap 通过将每个实例连接到最近的邻居来创建图形，然后在尝试保持实例之间的测地距离时降低维度。

- t-分布随机邻域嵌入（t-Distributed Stochastic Neighbor Embedding，t-SNE）可以用于降低维度，同时试图保持相似的实例临近并将不相似的实例分开。它主要用于可视化，尤其是用于可视化高维空间中的实例（例如，可以将MNIST图像降维到 2D 可视化）。
- 线性判别分析（Linear Discriminant Analysis，LDA）实际上是一种分类算法，但在训练过程中，它会学习类之间最有区别的轴，然后使用这些轴来定义用于投影数据的超平面。LDA 的好处是投影会尽可能地保持各个类之间距离，所以在运行另一种分类算法（如 SVM 分类器）之前，LDA 是很好的降维技术。

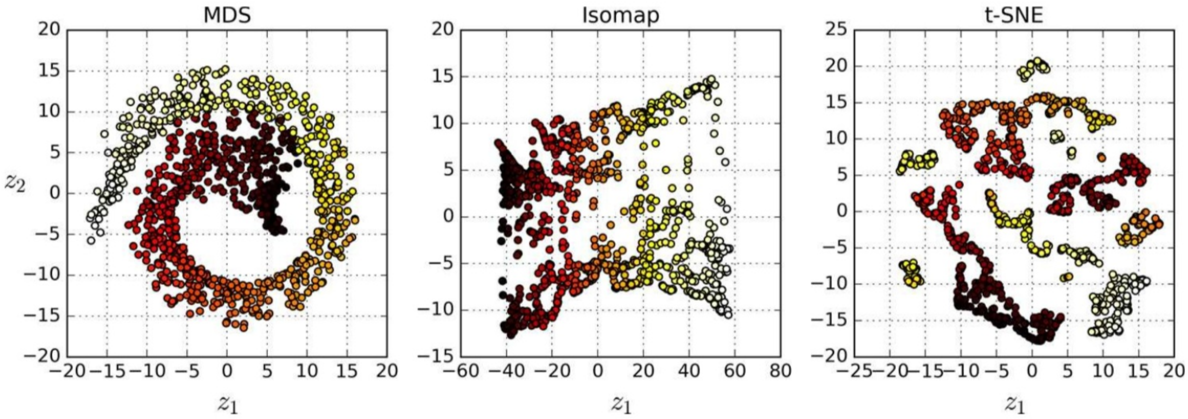


图 8-13 使用不同的技术将瑞士卷降维至 2D

练习

1. 减少数据集维度的主要动机是什么？主要缺点是什么？
2. 什么是维度爆炸？
3. 一旦对某数据集降维，我们可能恢复它吗？如果可以，怎样做才能恢复？如果不可以，为什么？
4. PCA 可以用于降低一个高度非线性对数据集吗？
5. 假设你对一个 1000 维的数据集应用 PCA，同时设置方差解释率为 95%，你的最终数据集将会有多少维？
6. 在什么情况下你会使用普通的 PCA，增量 PCA，随机 PCA 和核 PCA？
7. 你该如何评价你的降维算法在你数据集上的表现？
8. 将两个不同的降维算法串联使用有意义吗？
9. 加载 MNIST 数据集（在第 3 章中介绍），并将其分成一个训练集和一个测试集（将前 60,000 个实例用于训练，其余 10,000 个用于测试）。在数据集上训练一个随机森林分类器，并记录了花费多长时间，然后在测试集上评估模型。接下来，使用 PCA 降低数据集的维度，设置方差解释率为 95%。在降维后的数据集上训练一个新的随机森林分类器，并查看需要多长时间。训练速度更快？接下来评估测试集上的分类器：它与以前的分类器比较起来如何？
10. 使用 t-SNE 将 MNIST 数据集缩减到二维，并使用 Matplotlib 绘制结果图。您可以使用 10 种不同颜色的散点图来表示每个图像的目标类别。或者，您可以在每个实例的位置写

入彩色数字，甚至可以绘制数字图像本身的降维版本（如果绘制所有数字，则可视化可能会过于混乱，因此您应该绘制随机样本或只在周围没有其他实例被绘制的情况下绘制）。你将会得到一个分隔良好的的可视化数字集群。尝试使用其他降维算法，如 PCA，LLE 或 MDS，并比较可视化结果。

练习答案请见附录 A。

第9章 启动并运行TensorFlow

TensorFlow 是一款用于数值计算的强大的开源软件库，特别适用于大规模机器学习的微调。它的基本原理很简单：首先在 Python 中定义要执行的计算图（例如图 9-1），然后 TensorFlow 使用该图并使用优化的 C++ 代码高效运行该图。

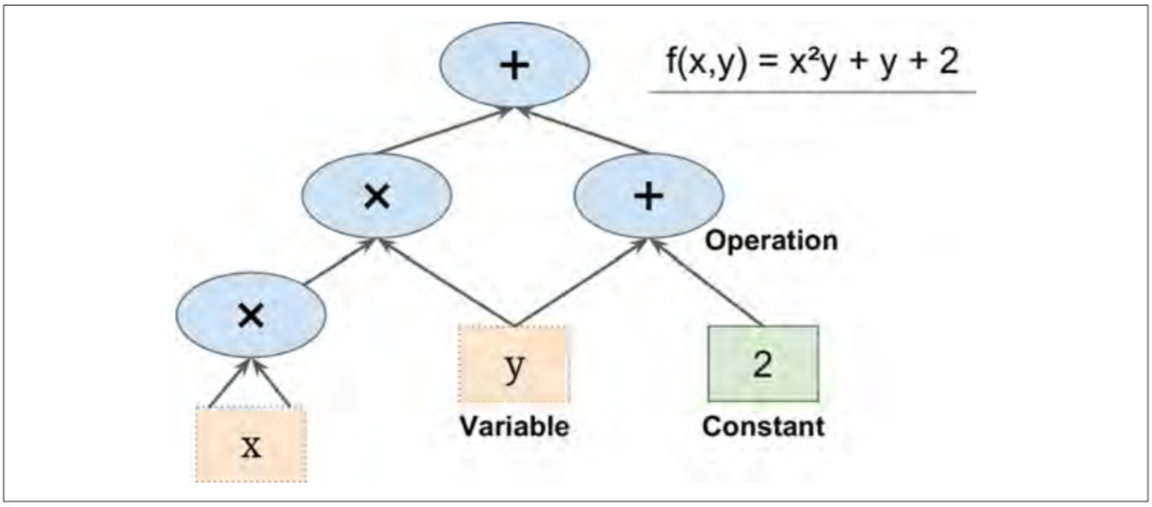


Figure 9-1. A simple computation graph

最重要的是，Tensorflow 可以将图分解为多个块并在多个 CPU 或 GPU 上并行运行（如图 9-2 所示）。 TensorFlow 还支持分布式计算，因此您可以在数百台服务器上分割计算，从而在合理的时间内在庞大的训练集上训练庞大的神经网络（请参阅第 12 章）。 TensorFlow 可以训练一个拥有数百万个参数的网络，训练集由数十亿个具有数百万个特征的实例组成。 这应该不会让您吃惊，因为 TensorFlow 是 由Google 大脑团队开发的，它支持谷歌的大量服务，例如 Google Cloud Speech，Google Photos 和 Google Search。

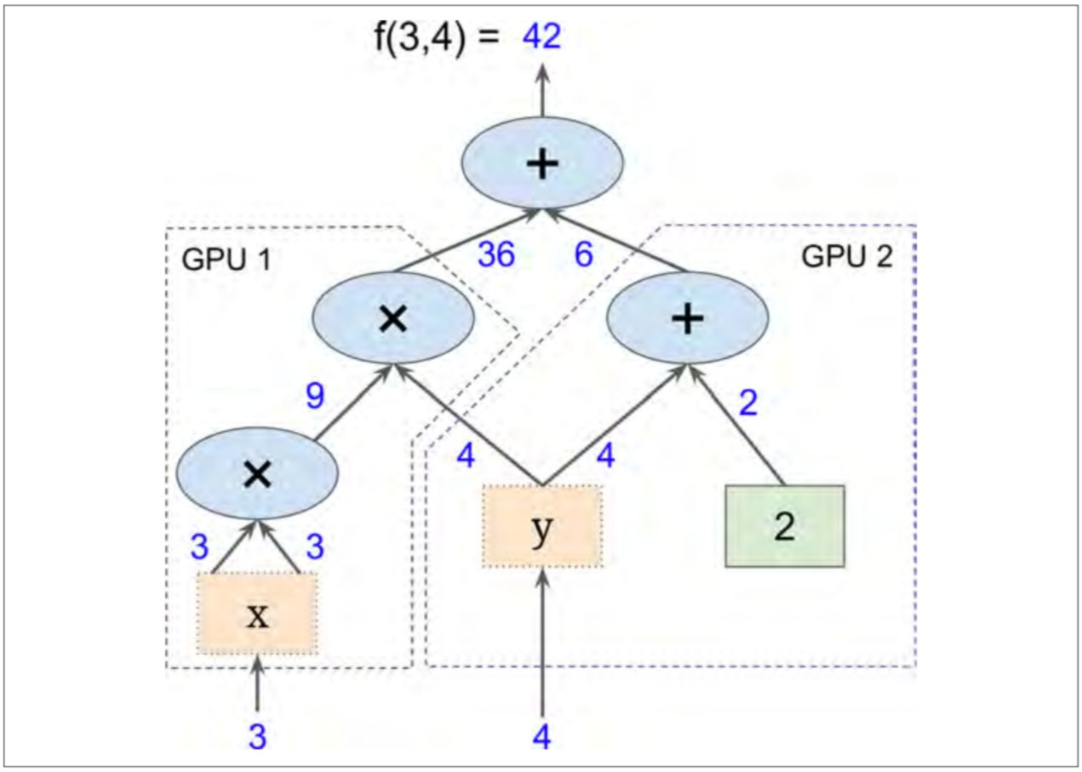


Figure 9-2. Parallel computation on multiple CPUs/GPUs/servers

当 TensorFlow 于 2015 年 11 月开放源代码时，已有许多深度学习的流行开源库（表 9-1 列出了一些），公平地说，大部分 TensorFlow 的功能已经存在于一个库或另一个库中。 尽管如此，TensorFlow 的整洁设计，可扩展性，灵活性和出色的文档（更不用说谷歌的名字）迅速将其推向了榜首。 简而言之，TensorFlow 的设计灵活性，可扩展性和生产就绪性，现有框架可以说只有其中三种可用。 这里有一些 TensorFlow 的亮点：

- 它不仅在 Windows，Linux 和 MacOS 上运行，而且在移动设备上运行，包括 iOS 和 Android。

  它提供了一个非常简单的 Python API，名为 TF.Learn2（`tensorflow.con trib.learn`），与 Scikit-Learn 兼容。正如你将会看到的，你可以用几行代码来训练不同类型的神经网络。之前是一个名为 Scikit Flow（或 Skow）的独立项目。

- 它还提供了另一个简单的称为 TF-slim（`tensorflow.contrib.slim`）的 API 来简化构建，训练和求出神经网络。
- 其他几个高级 API 已经在 TensorFlow 之上独立构建，如 **Keras** 或 **Pretty Tensor**。
- 它的主要 Python API 提供了更多的灵活性（以更高复杂度为代价）来创建各种计算，包括任何你能想到的神经网络结构。
- 它包括许多 ML 操作的高效 C ++ 实现，特别是构建神经网络所需的 C++ 实现。还有一个 C++ API 来定义您自己的高性能操作。

- 它提供了几个高级优化节点来搜索最小化损失函数的参数。由于 TensorFlow 自动处理计算您定义的函数的梯度，因此这些非常易于使用。这称为自动分解（或 `autodi` ）。
- 它还附带一个名为 TensorBoard 的强大可视化工具，可让您浏览计算图表，查看学习曲线等。
- Google 还推出了云服务来运行 TensorFlow 表。
- 最后，它拥有一支充满热情和乐于助人的开发团队，以及一个不断成长的社区，致力于改善它。它是 GitHub 上最受欢迎的开源项目之一，并且越来越多的优秀项目正在构建之上（例如，查看 https://www.tensorflow.org/ 或 https://github.com/jtoy/awesome-tensorflow）。要问技术问题，您应该使用 http://stackoverflow.com/ 并用 `tensorflow` 标记您的问题。您可以通过 GitHub 提交错误和功能请求。有关一般讨论，请加入 **Google** 小组。

Table 9-1. Open source Deep Learning libraries (not an exhaustive list)

Library	API	Platforms	Started by	Year
Caffe	Python, C++, Matlab	Linux, macOS, Windows	Y. Jia, UC Berkeley (BVLC)	2013
Deeplearning4j	Java, Scala, Clojure	Linux, macOS, Windows, Android	A. Gibson, J.Patterson	2014
H2O	Python, R	Linux, macOS, Windows	H2O.ai	2014
MXNet	Python, C++, others	Linux, macOS, Windows, iOS, Android	DMLC	2015
TensorFlow	Python, C++	Linux, macOS, Windows, iOS, Android	Google	2015
Theano	Python	Linux, macOS, iOS	University of Montreal	2010
Torch	C++, Lua	Linux, macOS, iOS, Android	R. Collobert, K. Kavukcuoglu, C. Farabet	2002

在本章中，我们将介绍 TensorFlow 的基础知识，从安装到创建，运行，保存和可视化简单的计算图。 在构建第一个神经网络之前掌握这些基础知识很重要（我们将在下一章中介绍）。

安装

让我们开始吧！假设您按照第 2 章中的安装说明安装了 Jupyter 和 Scikit-Learn，您可以简单地使用 `pip` 来安装 TensorFlow。 如果你使用 `virtualenv` 创建了一个独立的环境，你首先需要激活它：

```
$ cd $ML_PATH                                    #Your ML working
directory(e.g., $HOME/ml)
$ source env/bin/activate
```

下一步，安装 Tensorflow。

```
$ pip3 install --upgrade tensorflow
```

对于 GPU 支持，你需要安装 `tensorflow-gpu` 而不是 `tensorflow` 。具体请参见 12 章内容。

为了测试您的安装，请输入一下命令。其输出应该是您安装的 Tensorflow 的版本号。

```
$ python -c 'import tensorflow; print(tensorflow.__version__)'
1.0.0
```

创造第一个图谱，然后运行它

```
import tensorflow as tf
x = tf.Variable(3, name="x")
y = tf.Variable(4, name="y")
f = x*x*y + y + 2
```

这就是它的一切！ 最重要的是要知道这个代码实际上并不执行任何计算，即使它看起来像(尤其是最后一行）。 它只是创建一个计算图谱。 事实上，变量都没有初始化.要求出此图，您需要打开一个 TensorFlow 会话并使用它初始化变量并求出 `f` 。TensorFlow 会话负责处理在诸如 CPU 和 GPU 之类的设备上的操作并运行它们，并且它保留所有变量值。以下代码创建一个会话，初始化变量，并求出 `f` ，然后关闭会话（释放资源）：

```
# way1
sess = tf.Session()
sess.run(x.initializer)
sess.run(y.initializer)
result = sess.run(f)

print(result)
sess.close()
```

不得不每次重复sess.run() 有点麻烦，但幸运的是有一个更好的方法：

```
# way2
with tf.Session() as sess:
    x.initializer.run()
    y.initializer.run()
    result = f.eval()
print(result)
```

在 `with` 块中，会话被设置为默认会话。 调用 `x.initializer.run()` 等效于调用 `tf.get_default_session().run(x.initial)` ， `f.eval()` 等效于调用 `tf.get_default_session().run(f)` 。 这使得代码更容易阅读。 此外，会话在块的末尾自动关闭。

你可以使用 `global_variables_initializer()` 函数，而不是手动初始化每个变量。 请注意，它实际上没有立即执行初始化，而是在图谱中创建一个当程序运行时所有变量都会初始化的节点：

```
# way3
# init = tf.global_variables_initializer()
# with tf.Session() as sess:
#     init.run()
#     result = f.eval()
6. # print(result)
```

在 Jupyter 内部或在 Python shell 中，您可能更喜欢创建一个 `InteractiveSession` 。与常规会话的唯一区别是，当创建 `InteractiveSession` 时，它将自动将其自身设置为默认会话，因此您不需要使用模块（但是您需要在完成后手动关闭会话）：

```
# way4
init = tf.global_variables_initializer()
sess = tf.InteractiveSession()
init.run()
result = f.eval()
print(result)
sess.close()
```

TensorFlow 程序通常分为两部分：第一部分构建计算图谱（这称为构造阶段），第二部分运行它（这是执行阶段）。建设阶段通常构建一个表示 ML 模型的计算图谱,然后对其进行训练,计算。执行阶段通常运行循环，重复地求出训练步骤（例如，每个小批次），逐渐改进模型参数。

管理图谱

您创建的任何节点都会自动添加到默认图形中：

```
>>> x1 = tf.Variable(1)
>>> x1.graph is tf.get_default_graph()
True
```

在大多数情况下，这是很好的，但有时您可能需要管理多个独立图形。您可以通过创建一个新的图形并暂时将其设置为一个块中的默认图形，如下所示：

```
>>> graph = tf.Graph()
>>> with graph.as_default():
... x2 = tf.Variable(2)
...
>>> x2.graph is graph
True
>>> x2.graph is tf.get_default_graph()
False
```

在 Jupyter（或 Python shell）中，通常在实验时多次运行相同的命令。因此，您可能会收到包含许多重复节点的默认图形。一个解决方案是重新启动 Jupyter 内核（或 Python shell），但是一个更方便的解决方案是通过运行 `tf.reset_default_graph()` 来重置默认图。

节点值的生命周期

求出节点时，TensorFlow 会自动确定所依赖的节点集，并首先求出这些节点。 例如，考虑以下代码：

```
# w = tf.constant(3)
# x = w + 2
# y = x + 5
# z = x * 3

# with tf.Session() as sess:
#     print(y.eval())
#     print(z.eval())
```

首先，这个代码定义了一个非常简单的图。然后，它启动一个会话并运行图来求出 `y` ：TensorFlow 自动检测到 `y` 取决于 `x` ，它取决于 `w` ，所以它首先求出 `w` ，然后 `x` ，然后 `y` ，并返回 `y` 的值。最后，代码运行图来求出 `z` 。同样，TensorFlow 检测到它必须首先求出 `w` 和 `x` 。重要的是要注意，它不会复用以前的w和x的求出结果。简而言之，前面的代码求出 `w` 和 `x` 两次。 所有节点值都在图运行之间删除，除了变量值，由会话跨图形运行维护（队列和读者也保持一些状态）。变量在其初始化程序运行时启动其生命周期，并且在会话关闭时结束。 如果要有效地求出 `y` 和 `z` ，而不像之前的代码那样求出 `w` 和 `x` 两次，那么您必须要求 TensorFlow 在一个图形运行中求出 `y` 和 `z` ，如下面的代码所示：

```
# with tf.Session() as sess:
#     y_val, z_val = sess.run([y, z])
#     print(y_val) # 10
#     print(z_val) # 15
```

在单进程 TensorFlow 中，多个会话不共享任何状态，即使它们复用同一个图（每个会话都有自己的每个变量的副本）。 在分布式 TensorFlow 中，变量状态存储在服务器上，而不是在会话中，因此多个会话可以共享相同的变量。

Linear Regression with TensorFlow

TensorFlow 操作（也简称为 ops）可以采用任意数量的输入并产生任意数量的输出。 例如，加法运算和乘法运算都需要两个输入并产生一个输出。 常量和变量不输入（它们被称为源操作）。 输入和输出是称为张量的多维数组（因此称为“tensor flow”）。 就像 NumPy 数组一样，张量具有类型和形状。 实际上，在 Python API 中，张量简单地由 NumPy `ndarray` 表示。 它们通常包含浮点数，但您也可以使用它们来传送字符串（任意字节数组）。

迄今为止的示例，张量只包含单个标量值，但是当然可以对任何形状的数组执行计算。例如，以下代码操作二维数组来对加利福尼亚房屋数据集进行线性回归（在第 2 章中介绍）。它从获取数据集开始；之后它会向所有训练实例添加一个额外的偏置输入特征（ `x0 = 1` ）（它使用 NumPy 进行，因此立即运行）；之后它创建两个 TensorFlow 常量节点 `X` 和 `y` 来保存该数据和目标，并且它使用 TensorFlow 提供的一些矩阵运算来定义 `theta` 。这些矩阵函

数 `transpose()` ， `matmul()` 和 `matrix_inverse()` 是不言自明的，但是像往常一样，它们不会立即执行任何计算；相反，它们会在图形中创建在运行图形时执行它们的节点。您可以认识到 `θ` 的定义对应于方程 $(\hat{\theta} = \mathbf{X}^T \cdot \mathbf{X})^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$。

最后，代码创建一个 `session` 并使用它来求出 `theta` 。

```
import numpy as np
from sklearn.datasets import fetch_california_housing
housing = fetch_california_housing()
m, n = housing.data.shape
#np.c_按colunm来组合array
housing_data_plus_bias = np.c_[np.ones((m, 1)), housing.data]

X = tf.constant(housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")
XT = tf.transpose(X)
theta = tf.matmul(tf.matmul(tf.matrix_inverse(tf.matmul(XT, X)), XT), y)
with tf.Session() as sess:
    theta_value = theta.eval()
print(theta_value)
```

如果您有一个 GPU 的话，上述代码相较于直接使用 NumPy 计算正态方程式的主要优点是 TensorFlow 会自动运行在您的 GPU 上（如果您安装了支持 GPU 的 TensorFlow，则 TensorFlow 将自动运行在 GPU 上，请参阅第 12 章了解更多详细信息）。

其实这里就是用最小二乘法算 `θ`

http://blog.csdn.net/akon_wang_hkbu/article/details/77503725

实现梯度下降

让我们尝试使用批量梯度下降（在第 4 章中介绍），而不是普通方程。首先，我们将通过手动计算梯度来实现，然后我们将使用 TensorFlow 的自动扩展功能来使 TensorFlow 自动计算梯度，最后我们将使用几个 TensorFlow 的优化器。

当使用梯度下降时，请记住，首先要对输入特征向量进行归一化，否则训练可能要慢得多。您可以使用 TensorFlow，NumPy，Scikit-Learn 的 `StandardScaler` 或您喜欢的任何其他解决方案。以下代码假定此规范化已经完成。

手动计算渐变

以下代码应该是相当不言自明的，除了几个新元素：

- `random_uniform()` 函数在图形中创建一个节点，它将生成包含随机值的张量，给定其形状和值作用域，就像 NumPy 的 `rand()` 函数一样。
- `assign()` 函数创建一个为变量分配新值的节点。在这种情况下，它实现了批次梯度下降步骤 $\theta(nextstep) = \theta - \eta\nabla_\theta MSE(\theta)$。

- 主循环一次又一次（共 `n_epochs` 次）执行训练步骤，每 100 次迭代都打印出当前均方误差（MSE）。你应该看到 MSE 在每次迭代中都会下降。

```
housing = fetch_california_housing()
m, n = housing.data.shape
m, n = housing.data.shape
#np.c_按colunm来组合array
housing_data_plus_bias = np.c_[np.ones((m, 1)), housing.data]
scaled_housing_data_plus_bias = scale(housing_data_plus_bias)
n_epochs = 1000
learning_rate = 0.01
X = tf.constant(scaled_housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0), name="theta")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
gradients = 2/m * tf.matmul(tf.transpose(X), error)
training_op = tf.assign(theta, theta - learning_rate * gradients)
init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)
    for epoch in range(n_epochs):
        if epoch % 100 == 0:
            print("Epoch", epoch, "MSE =", mse.eval())
        sess.run(training_op)
    best_theta = theta.eval()
```

Using autodiff

前面的代码工作正常，但它需要从代价函数（MSE）中利用数学公式推导梯度。在线性回归的情况下，这是相当容易的，但是如果你必须用深层神经网络来做这个事情，你会感到头痛：这将是乏味和容易出错的。您可以使用符号求导来为您自动找到偏导数的方程式，但结果代码不一定非常有效。

为了理解为什么，考虑函数 `f(x) = exp(exp(exp(x)))` 。如果你知道微积分，你可以计算出它的导数 `f'(x) = exp(x) * exp(exp(x)) * exp(exp(exp(x)))` 。如果您按照普通的计算方式分别去写 `f(x)` 和 `f'(x)` ，您的代码将不会如此有效。一个更有效的解决方案是写一个首先计算 `exp(x)` ，然后 `exp(exp(x))` ，然后 `exp(exp(exp(x)))` 的函数，并返回所有三个。这直接给你（第三项） `f(x)` ，如果你需要求导，你可以把这三个子式相乘，你就完成了。通过传统的方法，您不得不将 `exp` 函数调用 9 次来计算 `f(x)` 和 `f'(x)` 。使用这种方法，你只需要调用它三次。

当您的功能由某些任意代码定义时，它会变得更糟。你能找到方程（或代码）来计算以下函数的偏导数吗？

提示：不要尝试。

```
def my_func(a, b):
    z = 0
    for i in range(100):
        z = a * np.cos(z + i) + z * np.sin(b - i)
    return z
```

幸运的是，TensorFlow 的自动计算梯度功能可以计算这个公式：它可以自动高效地为您计算梯度。 只需用以下面这行代码替换上一节中代码的 `gradients = ...` 行，代码将继续工作正常：

```
gradients = tf.gradients(mse, [theta])[0]
```

`gradients()` 函数使用一个 `op` （在这种情况下是MSE）和一个变量列表（在这种情况下只是 `theta` ），它创建一个 `ops` 列表（每个变量一个）来计算 `op` 的梯度变量。 因此，梯度节点将计算 MSE 相对于 `theta` 的梯度向量。

自动计算梯度有四种主要方法。 它们总结在表 9-2 中。 TensorFlow 使用反向模式，这是完美的（高效和准确），当有很多输入和少量的输出，如通常在神经网络的情况。 它只需要通过 $n_{outputs}+1$ 次图遍历即可计算所有输出的偏导数。

Table 9-2. Main solutions to compute gradients automatically

Technique	Nb of graph traversals to compute all gradients	Accuracy	Supports arbitrary code	Comment
Numerical differentiation	$n_{inputs}+1$	Low	Yes	Trivial to implement
Symbolic differentiation	N/A	High	No	Builds a very different graph
Forward-mode autodiff	n_{inputs}	High	Yes	Uses *dual numbers*
Reverse-mode autodiff	$n_{outputs}+1$	High	Yes	Implemented by TensorFlow

使用优化器

所以 TensorFlow 为您计算梯度。 但它还有更好的方法：它还提供了一些可以直接使用的优化器，包括梯度下降优化器。您可以使用以下代码简单地替换以前的 `gradients = ...` 和 `training_op = ...` 行，并且一切都将正常工作：

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(mse)
```

如果要使用其他类型的优化器，则只需要更改一行。 例如，您可以通过定义优化器来使用动量优化器（通常会比渐变收敛的收敛速度快得多；参见第 11 章）

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate, momentum=0.9)
```

将数据提供给训练算法

我们尝试修改以前的代码来实现小批量梯度下降（Mini-batch Gradient Descent）。为此，我们需要一种在每次迭代时用下一个小批量替换 `X` 和 `Y` 的方法。最简单的方法是使用占位符（placeholder）节点。这些节点是特别的，因为它们实际上并不执行任何计算，只是输出您在运行时输出的数据。它们通常用于在训练期间将训练数据传递给 TensorFlow。如果在运行时没有为占位符指定值，则会收到异常。

要创建占位符节点，您必须调用 `placeholder()` 函数并指定输出张量的数据类型。或者，您还可以指定其形状，如果要强制执行。如果指定维度为 `None`，则表示"任何大小"。例如，以下代码创建一个占位符节点 `A`，还有一个节点 `B = A + 5`。当我们求出 `B` 时，我们将一个 `feed_dict` 传递给 `eval()` 方法并指定 `A` 的值。注意，`A` 必须具有 2 级（即它必须是二维的），并且必须有三列（否则引发异常），但它可以有任意数量的行。

```
>>> A = tf.placeholder(tf.float32, shape=(None, 3))
>>> B = A + 5
>>> with tf.Session() as sess:
... B_val_1 = B.eval(feed_dict={A: [[1, 2, 3]]})
... B_val_2 = B.eval(feed_dict={A: [[4, 5, 6], [7, 8, 9]]})
...
>>> print(B_val_1)
[[ 6. 7. 8.]]
>>> print(B_val_2)
[[ 9. 10. 11.]
[ 12. 13. 14.]]
```

您实际上可以提供任何操作的输出，而不仅仅是占位符。在这种情况下，TensorFlow 不会尝试求出这些操作；它使用您提供的值。

要实现小批量渐变下降，我们只需稍微调整现有的代码。首先更改 `X` 和 `Y` 的定义，使其定义为占位符节点：

```
X = tf.placeholder(tf.float32, shape=(None, n + 1), name="X")
y = tf.placeholder(tf.float32, shape=(None, 1), name="y")
```

然后定义批量大小并计算总批次数：

```
batch_size = 100
n_batches = int(np.ceil(m / batch_size))
```

最后，在执行阶段，逐个获取小批量，然后在求出依赖于 `X` 和 `y` 的值的任何一个节点时，通过 `feed_dict` 提供 `X` 和 `y` 的值。

```
def fetch_batch(epoch, batch_index, batch_size):
    [...] # load the data from disk
    return X_batch, y_batch

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        for batch_index in range(n_batches):
            X_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
    best_theta = theta.eval()
```

在求出theta时，我们不需要传递X和y的值，因为它不依赖于它们。

MINI-BATCH 完整代码

```
import numpy as np
from sklearn.datasets import fetch_california_housing
import tensorflow as tf
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()
m, n = housing.data.shape
print("数据集:{}行,{}列".format(m,n))
housing_data_plus_bias = np.c_[np.ones((m, 1)), housing.data]
scaler = StandardScaler()
scaled_housing_data = scaler.fit_transform(housing.data)
scaled_housing_data_plus_bias = np.c_[np.ones((m, 1)), scaled_housing_data]

n_epochs = 1000
learning_rate = 0.01

X = tf.placeholder(tf.float32, shape=(None, n + 1), name="X")
y = tf.placeholder(tf.float32, shape=(None, 1), name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0, seed=42), name="theta")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(mse)

init = tf.global_variables_initializer()

n_epochs = 10
batch_size = 100
n_batches = int(np.ceil(m / batch_size)) # ceil() 方法返回 x 的值上限 - 不小于 x 的最小整数
。

def fetch_batch(epoch, batch_index, batch_size):
    know = np.random.seed(epoch * n_batches + batch_index)  # not shown in the book
    print("我是know:",know)
    indices = np.random.randint(m, size=batch_size)  # not shown
    X_batch = scaled_housing_data_plus_bias[indices] # not shown
    y_batch = housing.target.reshape(-1, 1)[indices] # not shown
    return X_batch, y_batch

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        for batch_index in range(n_batches):
            X_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})

    best_theta = theta.eval()

print(best_theta)
```

保存和恢复模型

一旦你训练了你的模型，你应该把它的参数保存到磁盘，所以你可以随时随地回到它，在另一个程序中使用它，与其他模型比较，等等。 此外，您可能希望在训练期间定期保存检查点，以便如果您的计算机在训练过程中崩溃，您可以从上次检查点继续进行，而不是从头开始。

TensorFlow 可以轻松保存和恢复模型。 只需在构造阶段结束（创建所有变量节点之后）创建一个保存节点; 那么在执行阶段，只要你想保存模型，只要调用它的 `save()` 方法:

```
[...]
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0), name="theta")
[...]
init = tf.global_variables_initializer()
saver = tf.train.Saver()
with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        if epoch % 100 == 0: # checkpoint every 100 epochs
            save_path = saver.save(sess, "/tmp/my_model.ckpt")

        sess.run(training_op)
    best_theta = theta.eval()
    save_path = saver.save(sess, "/tmp/my_model_final.ckpt")
```

恢复模型同样容易：在构建阶段结束时创建一个保存器，就像之前一样，但是在执行阶段的开始，而不是使用 `init` 节点初始化变量，你可以调用 `restore()` 方法 的保存器对象：

```
with tf.Session() as sess:
    saver.restore(sess, "/tmp/my_model_final.ckpt")
    [...]
```

默认情况下，保存器将以自己的名称保存并还原所有变量，但如果需要更多控制，则可以指定要保存或还原的变量以及要使用的名称。 例如，以下保存器将仅保存或恢复 `theta` 变量，它的键名称是 `weights` ：

```
saver = tf.train.Saver({"weights": theta})
```

完整代码

```
 numpy as np
from sklearn.datasets import fetch_california_housing
import tensorflow as tf
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()
m, n = housing.data.shape
print("数据集:{}行,{}列".format(m,n))
housing_data_plus_bias = np.c_[np.ones((m, 1)), housing.data]
scaler = StandardScaler()
scaled_housing_data = scaler.fit_transform(housing.data)
scaled_housing_data_plus_bias = np.c_[np.ones((m, 1)), scaled_housing_data]

n_epochs = 1000  # not shown in the book
learning_rate = 0.01  # not shown

X = tf.constant(scaled_housing_data_plus_bias, dtype=tf.float32, name="X")  # not shown

y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")  # not shown

theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0, seed=42), name="theta")
y_pred = tf.matmul(X, theta, name="predictions")  # not shown
error = y_pred - y  # not shown
mse = tf.reduce_mean(tf.square(error), name="mse")  # not shown
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)  # not shown

training_op = optimizer.minimize(mse)  # not shown

init = tf.global_variables_initializer()
saver = tf.train.Saver()

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        if epoch % 100 == 0:
            print("Epoch", epoch, "MSE =", mse.eval())  # not shown
            save_path = saver.save(sess, "/tmp/my_model.ckpt")
        sess.run(training_op)

    best_theta = theta.eval()
    save_path = saver.save(sess, "/tmp/my_model_final.ckpt") #找到tmp文件夹就找到文件了
```

使用 TensorBoard 展现图形和训练曲线

所以现在我们有一个使用小批量梯度下降训练线性回归模型的计算图谱，我们正在定期保存检查点。 听起来很复杂，不是吗？ 然而，我们仍然依靠 `print()` 函数可视化训练过程中的进度。 有一个更好的方法：进入 TensorBoard。如果您提供一些训练统计信息，它将在您的网络浏览器中显示这些统计信息的良好交互式可视化（例如学习曲线）。 您还可以提供图形的定义，它将为您提供一个很好的界面来浏览它。 这对于识别图中的错误，找到瓶颈等是非常有用的。

第一步是调整程序，以便将图形定义和一些训练统计信息（例如，`training_error` （MSE））写入 TensorBoard 将读取的日志目录。 您每次运行程序时都需要使用不同的日志目录，否则 TensorBoard 将会合并来自不同运行的统计信息，这将会混乱可视化。 最简单的解决方案是在日志目录名称中包含时间戳。 在程序开头添加以下代码：

```
from datetime import datetime
now = datetime.utcnow().strftime("%Y%m%d%H%M%S")
root_logdir = "tf_logs"
logdir = "{}/run-{}/".format(root_logdir, now)
```

接下来，在构建阶段结束时添加以下代码：

```
mse_summary = tf.summary.scalar('MSE', mse)
file_writer = tf.summary.FileWriter(logdir, tf.get_default_graph())
```

第一行创建一个节点，这个节点将求出 MSE 值并将其写入 TensorBoard 兼容的二进制日志字符串（称为摘要）中。 第二行创建一个 `FileWriter` ，您将用它来将摘要写入日志目录中的日志文件中。 第一个参数指示日志目录的路径（在本例中为 `tf_logs/run-20160906091959/` ，相对于当前目录）。 第二个（可选）参数是您想要可视化的图形。 创建时，文件写入器创建日志目录（如果需要），并将其定义在二进制日志文件（称为事件文件）中。

接下来，您需要更新执行阶段，以便在训练期间定期求出 `mse_summary` 节点（例如，每 10 个小批量）。 这将输出一个摘要，然后可以使用 `file_writer` 写入事件文件。 以下是更新的代码：

```
[...]
for batch_index in range(n_batches):
    X_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
    if batch_index % 10 == 0:
        summary_str = mse_summary.eval(feed_dict={X: X_batch, y: y_batch})
        step = epoch * n_batches + batch_index
        file_writer.add_summary(summary_str, step)
    sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
[...]
```

避免在每一个训练阶段记录训练数据，因为这会大大减慢训练速度（以上代码每 10 个小批量记录一次）.

最后，要在程序结束时关闭 `FileWriter` ：

```
file_writer.close()
```

完整代码

```
import numpy as np
from sklearn.datasets import fetch_california_housing
import tensorflow as tf
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()
m, n = housing.data.shape
print("数据集:{}行,{}列".format(m,n))
housing_data_plus_bias = np.c_[np.ones((m, 1)), housing.data]
scaler = StandardScaler()
scaled_housing_data = scaler.fit_transform(housing.data)
scaled_housing_data_plus_bias = np.c_[np.ones((m, 1)), scaled_housing_data]

from datetime import datetime

now = datetime.utcnow().strftime("%Y%m%d%H%M%S")
root_logdir = r"D://tf_logs"
logdir = "{}/run-{}/".format(root_logdir, now)
n_epochs = 1000
learning_rate = 0.01

X = tf.placeholder(tf.float32, shape=(None, n + 1), name="X")
y = tf.placeholder(tf.float32, shape=(None, 1), name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0, seed=42), name="theta")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(mse)

init = tf.global_variables_initializer()
mse_summary = tf.summary.scalar('MSE', mse)
file_writer = tf.summary.FileWriter(logdir, tf.get_default_graph())

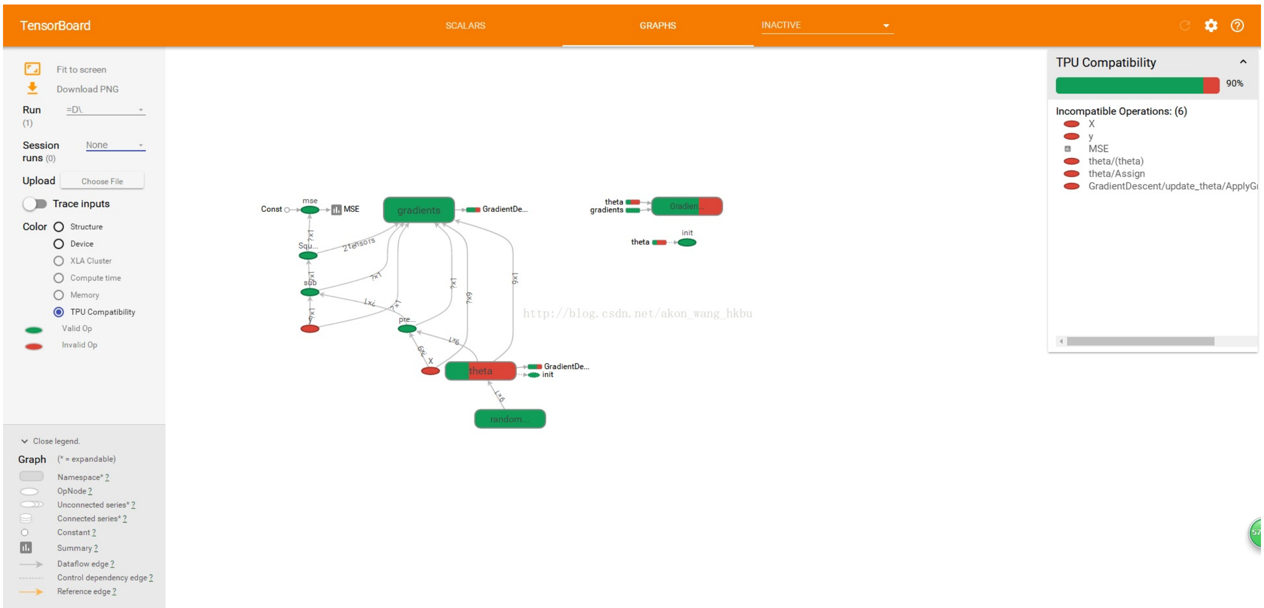
n_epochs = 10
batch_size = 100
n_batches = int(np.ceil(m / batch_size))

def fetch_batch(epoch, batch_index, batch_size):
    np.random.seed(epoch * n_batches + batch_index)  # not shown in the book
    indices = np.random.randint(m, size=batch_size)  # not shown
    X_batch = scaled_housing_data_plus_bias[indices] # not shown
    y_batch = housing.target.reshape(-1, 1)[indices] # not shown
    return X_batch, y_batch

with tf.Session() as sess:                                                        # no
t shown in the book
    sess.run(init)                                                                # no
t shown

    for epoch in range(n_epochs):                                                 # no
t shown
        for batch_index in range(n_batches):
            X_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
            if batch_index % 10 == 0:
                summary_str = mse_summary.eval(feed_dict={X: X_batch, y: y_batch})
                step = epoch * n_batches + batch_index
                file_writer.add_summary(summary_str, step)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})

    best_theta = theta.eval()
file_writer.close()
print(best_theta)
```



名称作用域

当处理更复杂的模型（如神经网络）时，该图可以很容易地与数千个节点混淆。为了避免这种情况，您可以创建名称作用域来对相关节点进行分组。例如，我们修改以前的代码来定义名为 `loss` 的名称作用域内的错误和 `mse` 操作：

```
with tf.name_scope("loss") as scope:
    error = y_pred - y
    mse = tf.reduce_mean(tf.square(error), name="mse")
```

在作用域内定义的每个 `op` 的名称现在以 `loss/` 为前缀：

```
>>> print(error.op.name)
loss/sub
>>> print(mse.op.name)
loss/mse
```

在 TensorBoard 中，`mse` 和 `error` 节点现在出现在 `loss` 命名空间中，默认情况下会出现崩溃（图 9-5）。

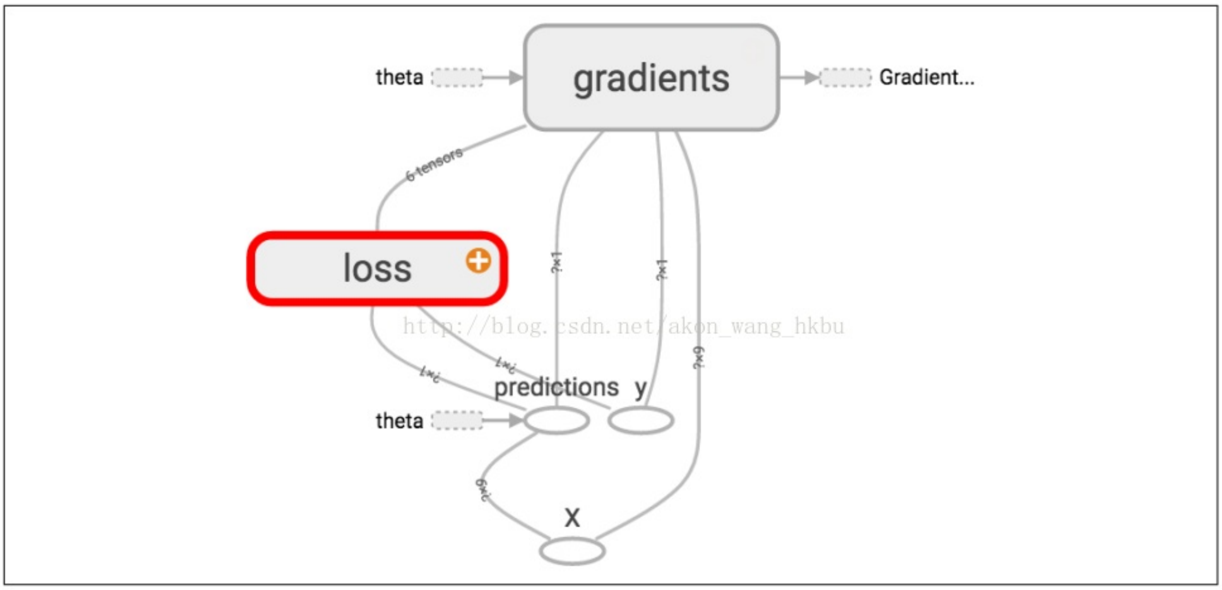


Figure 9-5. A collapsed namescope in TensorBoard

完整代码

```
import numpy as np
from sklearn.datasets import fetch_california_housing
import tensorflow as tf
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()
m, n = housing.data.shape
print("数据集:{}行,{}列".format(m,n))
housing_data_plus_bias = np.c_[np.ones((m, 1)), housing.data]
scaler = StandardScaler()
scaled_housing_data = scaler.fit_transform(housing.data)
scaled_housing_data_plus_bias = np.c_[np.ones((m, 1)), scaled_housing_data]

from datetime import datetime

now = datetime.utcnow().strftime("%Y%m%d%H%M%S")
root_logdir = r"D://tf_logs"
logdir = "{}/run-{}/".format(root_logdir, now)

n_epochs = 1000
learning_rate = 0.01

X = tf.placeholder(tf.float32, shape=(None, n + 1), name="X")
y = tf.placeholder(tf.float32, shape=(None, 1), name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0, seed=42), name="theta")
y_pred = tf.matmul(X, theta, name="predictions")

def fetch_batch(epoch, batch_index, batch_size):
    np.random.seed(epoch * n_batches + batch_index)  # not shown in the book
    indices = np.random.randint(m, size=batch_size)  # not shown
    X_batch = scaled_housing_data_plus_bias[indices] # not shown
    y_batch = housing.target.reshape(-1, 1)[indices] # not shown
    return X_batch, y_batch

with tf.name_scope("loss") as scope:
    error = y_pred - y
    mse = tf.reduce_mean(tf.square(error), name="mse")

optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(mse)

init = tf.global_variables_initializer()

mse_summary = tf.summary.scalar('MSE', mse)
file_writer = tf.summary.FileWriter(logdir, tf.get_default_graph())

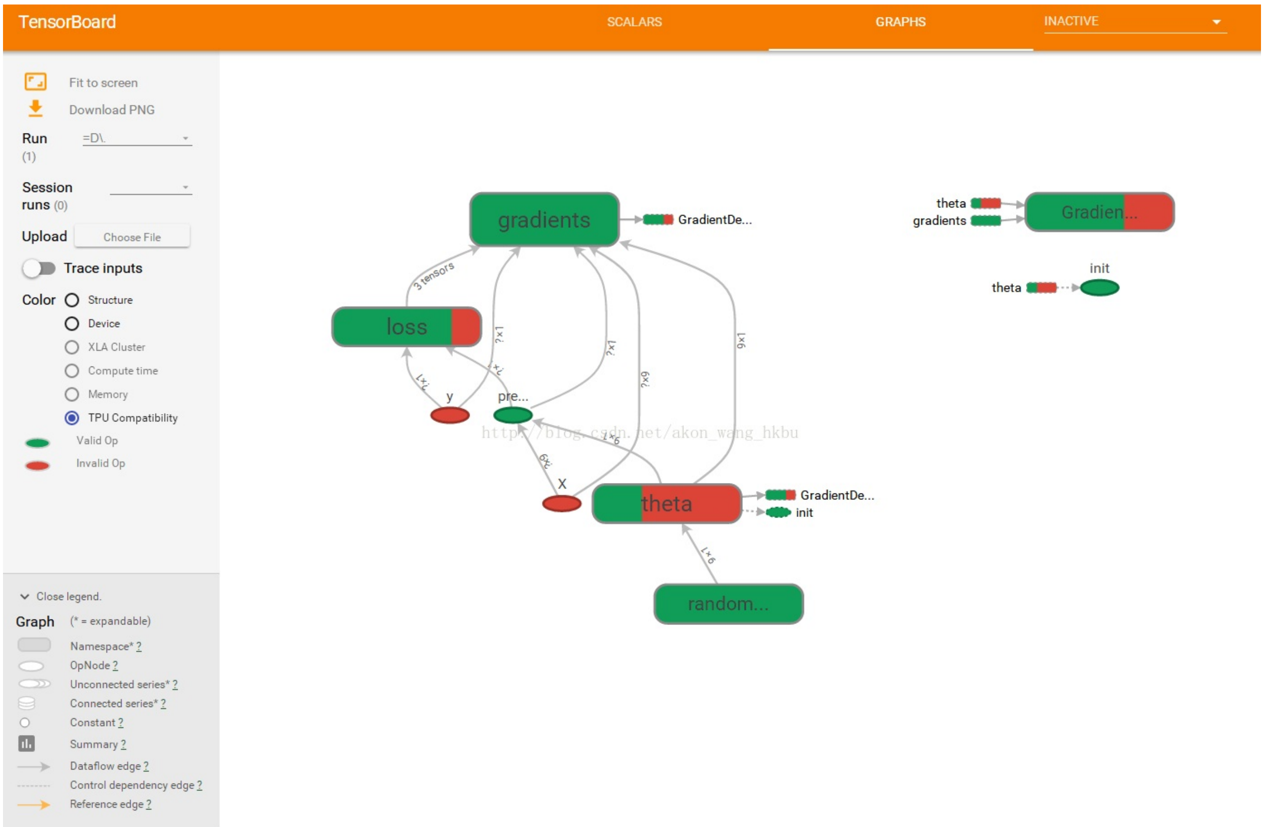
n_epochs = 10
batch_size = 100
n_batches = int(np.ceil(m / batch_size))

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        for batch_index in range(n_batches):
            X_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
            if batch_index % 10 == 0:
                summary_str = mse_summary.eval(feed_dict={X: X_batch, y: y_batch})
                step = epoch * n_batches + batch_index
                file_writer.add_summary(summary_str, step)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})

    best_theta = theta.eval()

file_writer.flush()
file_writer.close()
print("Best theta:")
print(best_theta)
```



模块性

假设您要创建一个图，它的作用是将两个整流线性单元（ReLU）的输出值相加。ReLU 计算一个输入值的对应线性函数输出值，如果为正，则输出该结值，否则为 0，如等式 9-1 所示。

Equation 9-1. Rectified linear unit

$$h_{\mathbf{w},b}(\mathbf{X}) = \max(\mathbf{X}\cdot\mathbf{w}+b, 0)$$

下面的代码做这个工作，但是它是相当重复的：

```
n_features = 3
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
w1 = tf.Variable(tf.random_normal((n_features, 1)), name="weights1")
w2 = tf.Variable(tf.random_normal((n_features, 1)), name="weights2")
b1 = tf.Variable(0.0, name="bias1")
b2 = tf.Variable(0.0, name="bias2")
z1 = tf.add(tf.matmul(X, w1), b1, name="z1")
z2 = tf.add(tf.matmul(X, w2), b2, name="z2")
relu1 = tf.maximum(z1, 0., name="relu1")
relu2 = tf.maximum(z1, 0., name="relu2")
output = tf.add(relu1, relu2, name="output")
```

这样的重复代码很难维护，容易出错（实际上，这个代码包含了一个剪贴错误，你发现了吗？） 如果你想添加更多的 ReLU，会变得更糟。 幸运的是，TensorFlow 可以让您保持 DRY（不要重复自己）：只需创建一个功能来构建 ReLU。 以下代码创建五个 ReLU 并输出其总和（注意，`add_n()` 创建一个计算张量列表之和的操作）：

```
def relu(X):
    w_shape = (int(X.get_shape()[1]), 1)
    w = tf.Variable(tf.random_normal(w_shape), name="weights")
    b = tf.Variable(0.0, name="bias")
    z = tf.add(tf.matmul(X, w), b, name="z")
    return tf.maximum(z, 0., name="relu")

n_features = 3
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
relus = [relu(X) for i in range(5)]
output = tf.add_n(relus, name="output")
```

请注意，创建节点时，TensorFlow 将检查其名称是否已存在，如果它已经存在，则会附加一个下划线，后跟一个索引，以使该名称是唯一的。 因此，第一个 ReLU 包含名为 `weights` ，`bias` ，`z` 和 `relu` 的节点（加上其他默认名称的更多节点，如 `MatMul` ）; 第二个 ReLU 包含名为 `weights_1` ，`bias_1` 等节点的节点; 第三个 ReLU 包含名为 `weights_2` ，`bias_2` 的节点，依此类推。 TensorBoard 识别这样的系列并将它们折叠在一起以减少混乱（如图 9-6 所示）

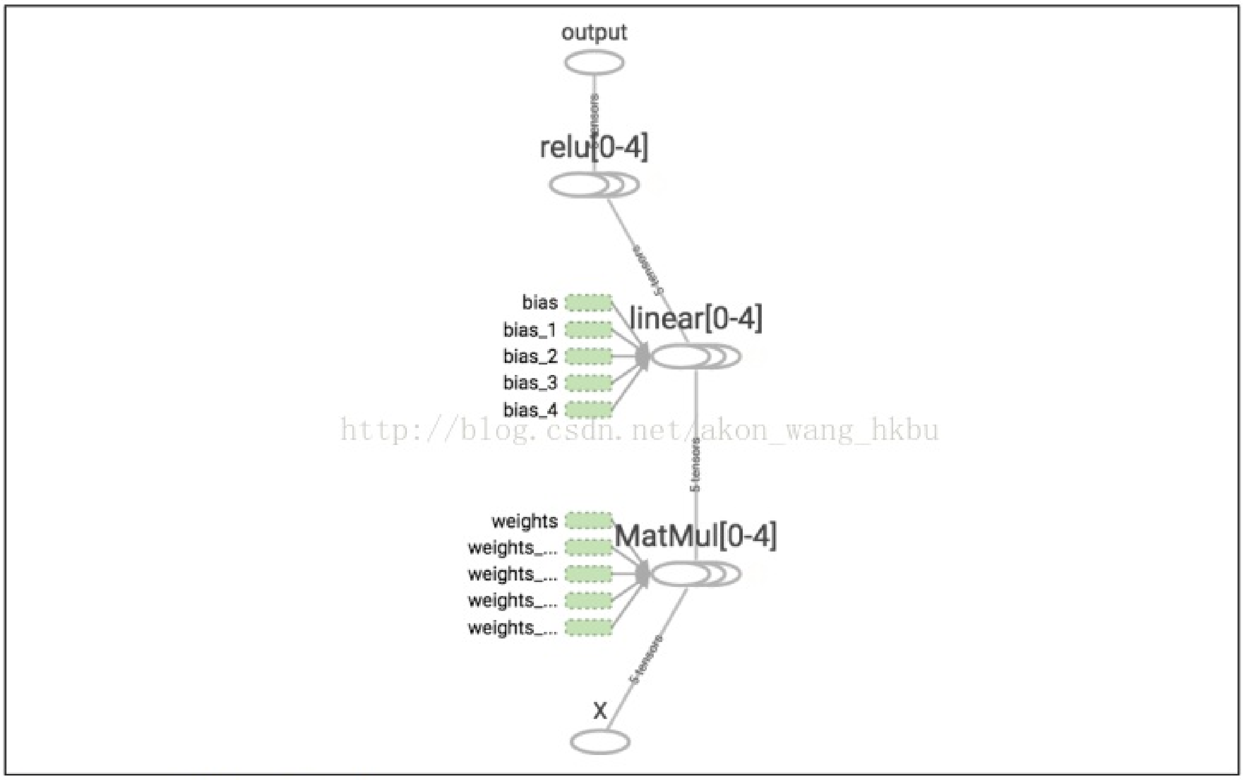


Figure 9-6. Collapsed node series

使用名称作用域，您可以使图形更清晰。 简单地将 `relu()` 函数的所有内容移动到名称作用域内。 图 9-7 显示了结果图。 请注意，TensorFlow 还通过附加 `_1` ，`_2` 等来提供名称作用域的唯一名称。

```
def relu(X):
    with tf.name_scope("relu"):
        w_shape = (int(X.get_shape()[1]), 1)                    # not shown in t
he book
        w = tf.Variable(tf.random_normal(w_shape), name="weights")    # not shown
        b = tf.Variable(0.0, name="bias")                             # not shown
        z = tf.add(tf.matmul(X, w), b, name="z")                      # not shown
        return tf.maximum(z, 0., name="max")                          # not shown
```

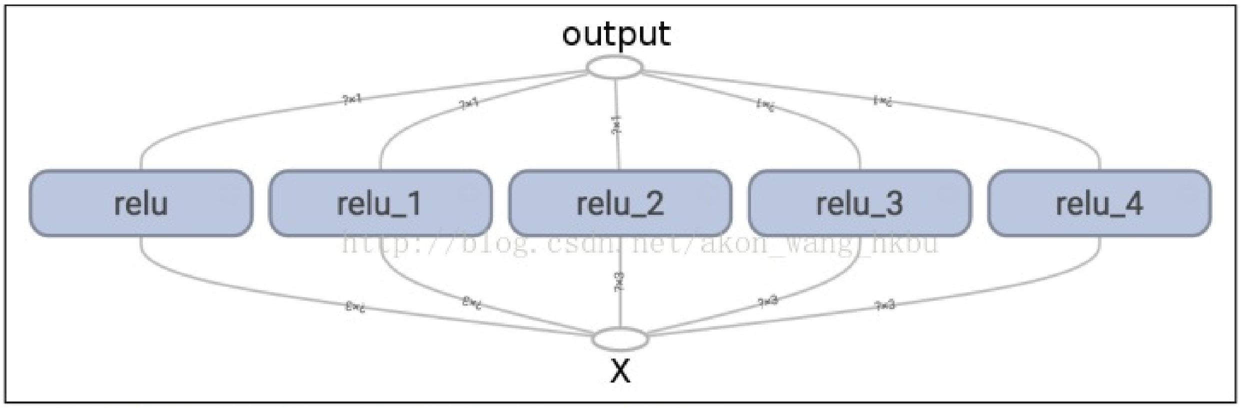


Figure 9-7. A clearer graph using name-scoped units

共享变量

如果要在图形的各个组件之间共享一个变量，一个简单的选项是首先创建它，然后将其作为参数传递给需要它的函数。 例如，假设要使用所有 ReLU 的共享阈值变量来控制 ReLU 阈值（当前硬编码为 0）。 您可以先创建该变量，然后将其传递给 `relu()` 函数：

```
reset_graph()

def relu(X, threshold):
    with tf.name_scope("relu"):
        w_shape = (int(X.get_shape()[1]), 1)                    # not shown in the
 book
        w = tf.Variable(tf.random_normal(w_shape), name="weights")  # not shown
        b = tf.Variable(0.0, name="bias")                           # not shown
        z = tf.add(tf.matmul(X, w), b, name="z")                    # not shown
        return tf.maximum(z, threshold, name="max")

threshold = tf.Variable(0.0, name="threshold")
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
relus = [relu(X, threshold) for i in range(5)]
output = tf.add_n(relus, name="output")
```

这很好:现在您可以使用阈值变量来控制所有 ReLU 的阈值。但是，如果有许多共享参数，比如这一项，那么必须一直将它们作为参数传递，这将是非常痛苦的。许多人创建了一个包含模型中所有变量的 Python 字典，并将其传递给每个函数。另一些则为每个模块创建一个类（例如：一个使用类变量来处理共享参数的 ReLU 类）。另一种选择是在第一次调用时将共享变量设置为 `relu()` 函数的属性，如下所示:

```
def relu(X):
    with tf.name_scope("relu"):
        if not hasattr(relu, "threshold"):
            relu.threshold = tf.Variable(0.0, name="threshold")
        w_shape = int(X.get_shape()[1]), 1                          # not shown in the
 book
        w = tf.Variable(tf.random_normal(w_shape), name="weights")  # not shown
        b = tf.Variable(0.0, name="bias")                           # not shown
        z = tf.add(tf.matmul(X, w), b, name="z")                    # not shown
        return tf.maximum(z, relu.threshold, name="max")
```

TensorFlow 提供了另一个选项，这将提供比以前的解决方案稍微更清洁和更模块化的代码。首先要明白一点，这个解决方案很刁钻难懂，但是由于它在 TensorFlow 中使用了很多，所以值得我们去深入细节。 这个想法是使用 `get_variable()` 函数来创建共享变量，如果它还不存在，或者如果已经存在，则复用它。 所需的行为（创建或复用）由当前 `variable_scope()` 的属性控制。 例如，以下代码将创建一个名为 `relu/threshold` 的变量（作为标量，因为 `shape = ()` ，并使用 0.0 作为初始值）：

```
with tf.variable_scope("relu"):
    threshold = tf.get_variable("threshold", shape=(),
                                initializer=tf.constant_initializer(0.0))
```

请注意，如果变量已经通过较早的 `get_variable()` 调用创建，则此代码将引发异常。 这种行为可以防止错误地复用变量。如果要复用变量，则需要通过将变量 `scope` 的复用属性设置为 `True` 来明确说明（在这种情况下，您不必指定形状或初始值）：

```
with tf.variable_scope("relu", reuse=True):
    threshold = tf.get_variable("threshold")
```

该代码将获取现有的 `relu/threshold` 变量，如果不存在会引发异常（如果没有使用 `get_variable()` 创建）。 或者，您可以通过调用 `scope` 的 `reuse_variables()` 方法将复用属性设置为 `true` ：

```
with tf.variable_scope("relu") as scope:
    scope.reuse_variables()
    threshold = tf.get_variable("threshold")
```

一旦重新使用设置为 `True` ，它将不能在块内设置为 `False` 。 而且，如果在其中定义其他变量作用域，它们将自动继承 `reuse = True` 。 最后，只有通过 `get_variable()` 创建的变量才可以这样复用.

现在，您拥有所有需要的部分，使 `relu()` 函数访问阈值变量，而不必将其作为参数传递：

```
def relu(X):
    with tf.variable_scope("relu", reuse=True):
        threshold = tf.get_variable("threshold")
        w_shape = int(X.get_shape()[1]), 1                          # not shown
        w = tf.Variable(tf.random_normal(w_shape), name="weights")  # not shown
        b = tf.Variable(0.0, name="bias")                           # not shown
        z = tf.add(tf.matmul(X, w), b, name="z")                    # not shown
        return tf.maximum(z, threshold, name="max")

X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
with tf.variable_scope("relu"):
    threshold = tf.get_variable("threshold", shape=(),
                                initializer=tf.constant_initializer(0.0))
relus = [relu(X) for relu_index in range(5)]
output = tf.add_n(relus, name="output")
```

该代码首先定义 `relu()` 函数，然后创建 `relu/threshold` 变量（作为标量，稍后将被初始化为 0.0），并通过调用 `relu()` 函数构建五个ReLU。`relu()` 函数复用 `relu/threshold` 变量，并创建其他 ReLU 节点。

使用 `get_variable()` 创建的变量始终以其 `variable_scope` 的名称作为前缀命名（例如，`relu/threshold`），但对于所有其他节点（包括使用 `tf.Variable()` 创建的变量），变量作用域的行为就像一个新名称的作用域。特别是，如果已经创建了具有相同名称的名称作用域，则添加后缀以使该名称是唯一的。例如，在前面的代码中创建的所有节点（阈值变量除外）的名称前缀为 `relu_1/` 到 `relu_5/`，如图 9-8 所示。

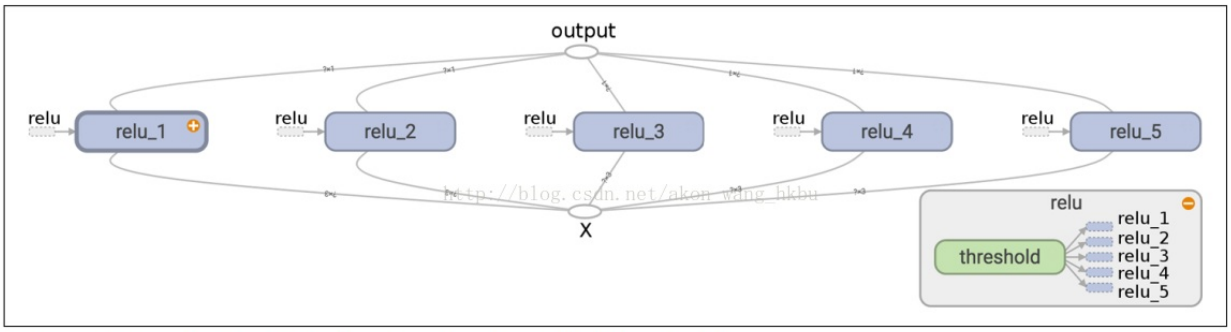


Figure 9-8. Five ReLUs sharing the threshold variable

不幸的是，必须在 `relu()` 函数之外定义阈值变量，其中 ReLU 代码的其余部分都驻留在其中。要解决此问题，以下代码在第一次调用时在 `relu()` 函数中创建阈值变量，然后在后续调用中重新使用。现在，`relu()` 函数不必担心名称作用域或变量共享：它只是调用 `get_variable()`，它将创建或复用阈值变量（它不需要知道是哪种情况）。其余的代码调用 `relu()` 五次，确保在第一次调用时设置 `reuse = False`，而对于其他调用来说，`reuse = True`。

```
def relu(X):
    threshold = tf.get_variable("threshold", shape=(),
                                initializer=tf.constant_initializer(0.0))
    w_shape = (int(X.get_shape()[1]), 1)                          # not shown in the book

    w = tf.Variable(tf.random_normal(w_shape), name="weights")  # not shown
    b = tf.Variable(0.0, name="bias")                           # not shown
    z = tf.add(tf.matmul(X, w), b, name="z")                    # not shown
    return tf.maximum(z, threshold, name="max")

X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
relus = []
for relu_index in range(5):
    with tf.variable_scope("relu", reuse=(relu_index >= 1)) as scope:
        relus.append(relu(X))
output = tf.add_n(relus, name="output")
```

生成的图形与之前略有不同，因为共享变量存在于第一个 ReLU 中（见图 9-9）。

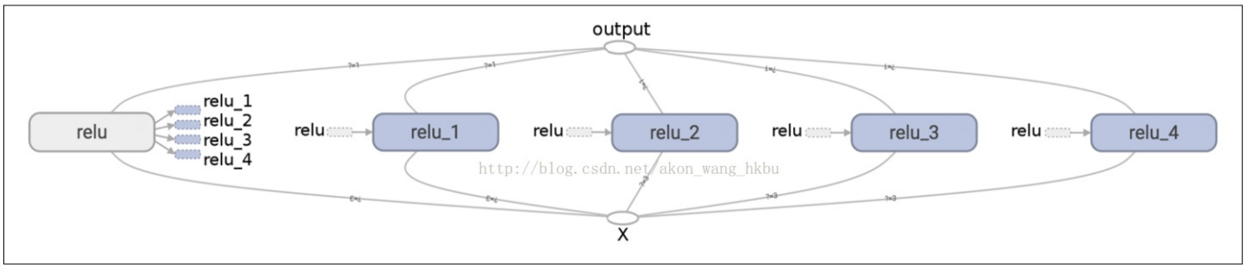


Figure 9-9. Five ReLUs sharing the threshold variable

TensorFlow 的这个介绍到此结束。我们将在以下章节中讨论更多高级课题，特别是与深层神经网络，卷积神经网络和递归神经网络相关的许多操作，以及如何使用多线程，队列，多个 GPU 以及如何将 TensorFlow 扩展到多台服务器。

第10章 人工神经网络介绍

鸟类启发我们飞翔，牛蒡植物启发了尼龙绳，大自然也激发了许多其他发明。从逻辑上看，大脑是如何构建智能机器的灵感。这是启发人工神经网络（ANN）的关键思想。然而，尽管飞机受到鸟类的启发，但它们不必拍动翅膀。同样的，ANN 逐渐变得与他们的生物表兄弟有很大的不同。一些研究者甚至争辩说，我们应该完全放弃生物类比（例如，通过说“单位”而不是“神经元”），以免我们把我们的创造力限制在生物学的系统上。

人工神经网络是深度学习的核心。它们具有通用性、强大性和可扩展性，使得它们能够很好地解决大型和高度复杂的机器学习任务，例如分类数十亿图像（例如，谷歌图像），强大的语音识别服务（例如，苹果的 Siri），通过每天追踪数百万的用户的行为推荐最好的视频（比如 YouTube），或者通过在游戏中击败世界冠军，通过学习数百万的游戏，然后与自己对抗（DeepMind 的 AlgFaGo）。

在本章中，我们将介绍人工神经网络，从快速游览的第一个ANN架构开始。然后，我们将提出多层感知器（MLP），并基于TensorFlow实现MNIST数字分类问题（在第3章中介绍）。

从生物到人工神经元

令人惊讶的是，人工神经网络已经存在了相当长的一段时间：它们最初是由神经生理学家 Warren McCulloch 和数学家 Walter Pitts 在 1943 提出。McCulloch 和 Pitts 在其里程碑式的论文中提出了“神经活动内在的逻辑演算”，提出了一个简化的计算模型，即生物神经元如何在动物大脑中协同工作，用逻辑进行复杂的计算。这是第一个人工神经网络体系结构。从那时起，正如我们将看到的，许多其他的神经元结构已经被发明，

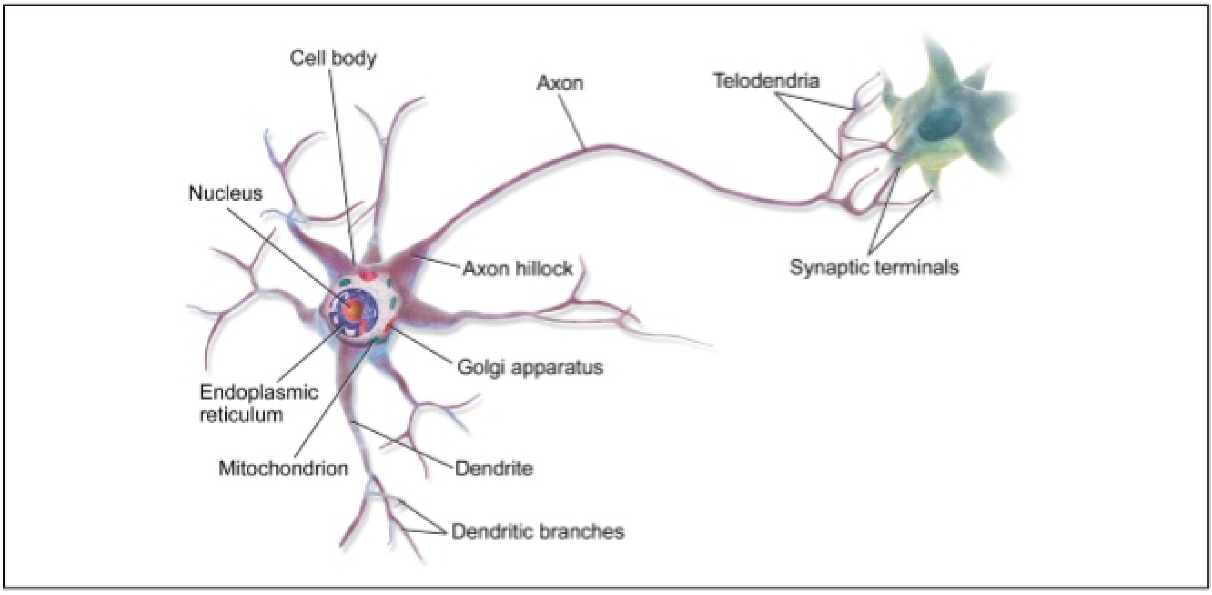
直到 20 世纪 60 年代，安纳斯的早期成功才使人们普遍相信我们很快就会与真正的智能机器对话。当显然的这个承诺将不会被兑现（至少相当长一段时间）时，资金飞向别处，ANN 进入了一个漫长的黑暗时代。20 世纪 80 年代初，随着新的网络体系结构的发明和更好的训练技术的发展，人们对人工神经网络的兴趣也在重新燃起。但到了 20 世纪 90 年代，强大的可替代机器学习技术的，如支持向量机（见第5章）受到大多数研究者的青睐，因为它们似乎提供了更好的结果和更强的理论基础。最后，我们现在目睹了另一股对 ANN 感兴趣的浪潮。这波会像以前一样消失吗？有一些很好的理由相信，这一点是不同的，将会对我们的生活产生更深远的影响：

- 现在有大量的数据可用于训练神经网络，ANN 在许多非常复杂的问题上经常优于其他 ML 技术。
- 自从 90 年代以来，计算能力的巨大增长使得在合理的时间内训练大型神经网络成为可能。这部分是由于穆尔定律，但也得益于游戏产业，它已经产生了数以百万计的强大的 GPU 显卡。

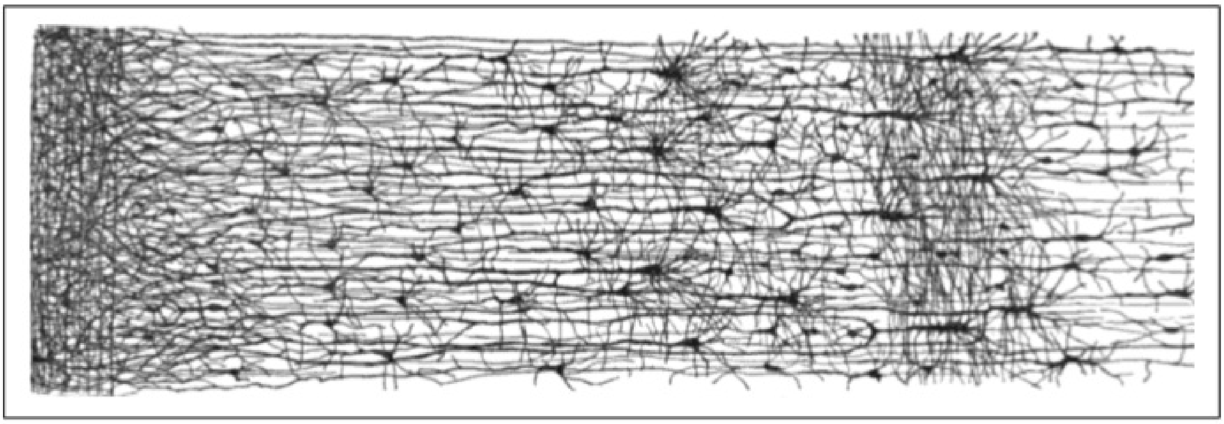
- 改进了训练算法。公平地说，它们与上世纪 90 年代使用的略有不同，但这些相对较小的调整产生了巨大的正面影响。
- 在实践中，人工神经网络的一些理论局限性是良性的。例如，许多人认为人工神经网络训练算法是注定的，因为它们很可能陷入局部最优，但事实证明，这在实践中是相当罕见的（或者如果它发生，它们也通常相当接近全局最优）。
- ANN 似乎已经进入了资金和进步的良性循环。基于 ANN 的惊人产品定期成为头条新闻，吸引了越来越多的关注和资金，导致越来越多的进步，甚至更惊人的产品。

生物神经元

在我们讨论人工神经元之前，让我们快速看一个生物神经元（如图 10-1 所示）。它是一种异常细胞，主要见于动物大脑皮层（例如，你的大脑），由包含细胞核和大多数细胞复杂成分的细胞体组成，许多分支扩展称为树突，加上一个非常长的延伸称为轴突。轴突的长度可能比细胞体长几倍，或长达几万倍。在它的末端附近，轴突分裂成许多称为 telodendria 的分支，在这些分支的顶端是微小的结构，称为突触末端（或简单的突触），它们连接到其他神经元的树突（或直接到细胞体）。生物神经元接收短的电脉冲，称为来自其他神经元的信号，通过这些突触。当神经元在几毫秒内接收到来自其他神经元的足够数量的信号时，它就发射出自己的信号。

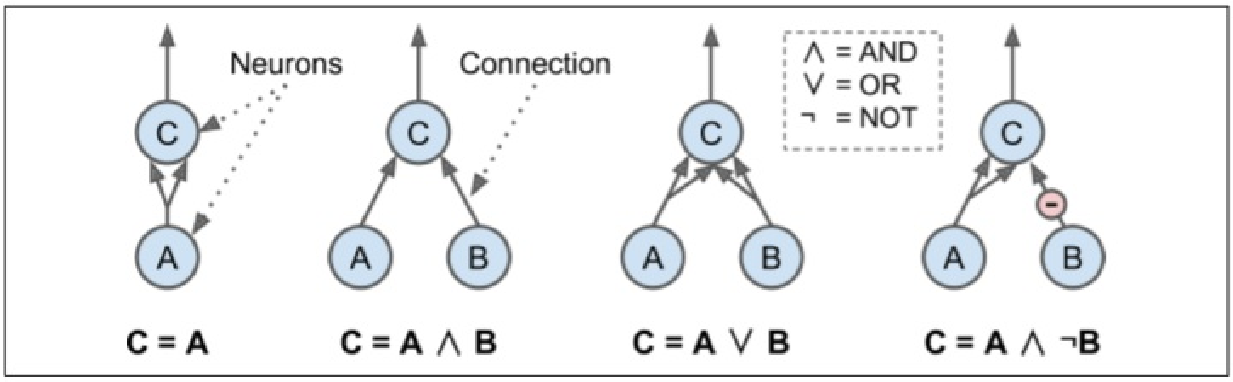


因此，个体的生物神经元似乎以一种相当简单的方式运行，但是它们组织在一个巨大的数十亿神经元的网络中，每个神经元通常连接到数千个其他神经元。高度复杂的计算可以由相当简单的神经元的巨大网络来完成，就像一个复杂的蚁穴可以由每个蚂蚁的努力构造出来。生物神经网络（BNN）的体系结构仍然是主动研究的主题，但是大脑的某些部分已经被映射，并且似乎神经元经常组织在连续的层中，如图 10-2 所示。



神经元的逻辑计算

Warren McCulloch 和 Pitts 提出一个非常简单的生物神经元模型，这后来作为一个人工神经元成为众所周知：它有一个或更多的二进制（ON/OFF）输入和一个二进制输出。当超过一定数量的输入是激活时，人工神经元会激活其输出。McCulloch 和 Pitts 表明，即使用这样一个简化的模型，也有可能建立一个人工神经元网络来计算任何你想要的逻辑命题。例如，让我们构建一些执行各种逻辑计算的 ANN（见图 10-3），假设当至少两个输入是激活的时候神经元被激活。

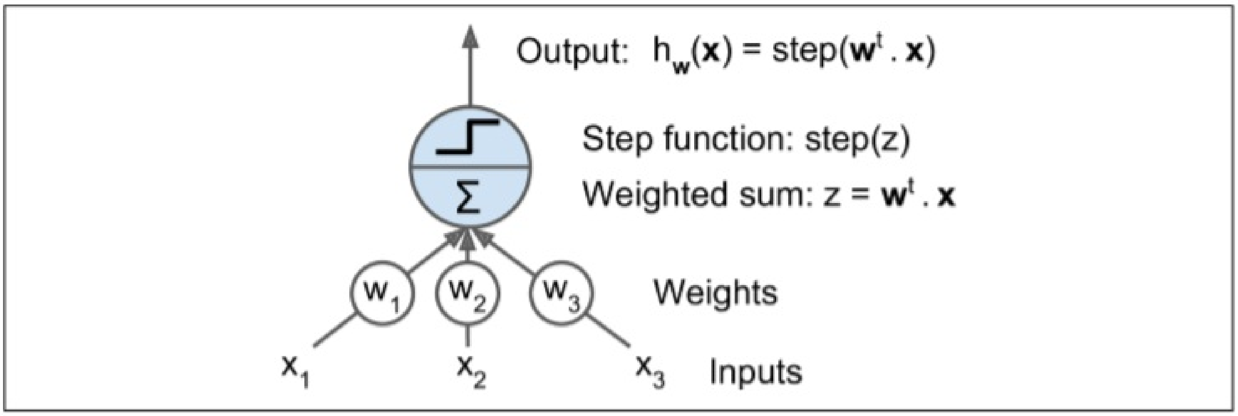


- 左边的第一个网络仅仅是确认函数：如果神经元 A 被激活，那么神经元 C 也被激活（因为它接收来自神经元 A 的两个输入信号），但是如果神经元 A 关闭，那么神经元 C 也关闭。
- 第二网络执行逻辑 AND：神经元 C 只有在激活神经元 A 和 B（单个输入信号不足以激活神经元 C）时才被激活。
- 第三网络执行逻辑 OR：如果神经元 A 或神经元 B 被激活（或两者），神经元 C 被激活。
- 最后，如果我们假设输入连接可以抑制神经元的活动（生物神经元是这样的情况），那么第四个网络计算一个稍微复杂的逻辑命题：如果神经元 B 关闭，只有当神经元A是激活的，神经元 C 才被激活。如果神经元 A 始终是激活的，那么你得到一个逻辑 NOT：神经元 C 在神经元 B 关闭时是激活的，反之亦然。

您可以很容易地想象如何将这些网络组合起来计算复杂的逻辑表达式（参见本章末尾的练习）。

感知器

感知器是最简单的人工神经网络结构之一，由 Frank Rosenblatt 发明于 1957。它是基于一种稍微不同的人工神经元（见图 10-4），称为线性阈值单元（LTU）：输入和输出现在是数字（而不是二进制开/关值），并且每个输入连接都与权重相连。LTU计算其输入的加权和（ `z = W1×1 + W2×2 + ... + + WN×n = Wt·x` ），然后将阶跃函数应用于该和，并输出结果： `HW(x) = STEP(Z) = STEP(W^T·x)` 。



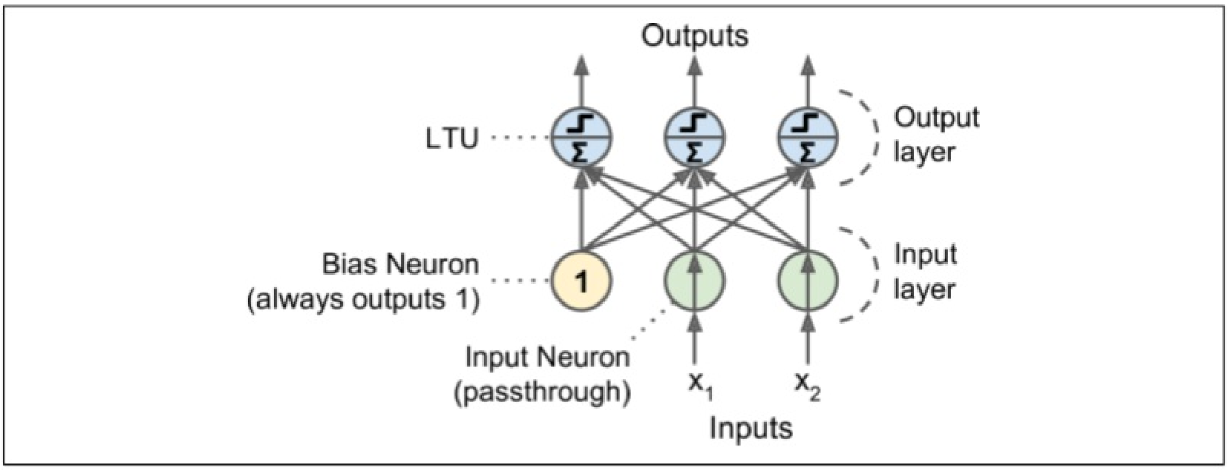
最常见的在感知器中使用的阶跃函数是 Heaviside 阶跃函数（见方程 10-1）。有时使用符号函数代替。

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \qquad \text{sgn}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

单一的 LTU 可被用作简单线性二元分类。它计算输入的线性组合，如果结果超过阈值，它输出正类或者输出负类（就像一个逻辑回归分类或线性 SVM）。例如，你可以使用单一的 LTU 基于花瓣长度和宽度去分类鸢尾花（也可添加额外的偏置特征 `x0=1` ，就像我们在前一章所做的）。训练一个 LTU 意味着去寻找合适的 `W0` 和 `W1` 值，（训练算法稍后提到）。

感知器简单地由一层 LTU 组成，每个神经元连接到所有输入。这些连接通常用特殊的被称为输入神经元的传递神经元来表示：它们只输出它们所输入的任何输入。此外，通常添加额外偏置特征（ `X0=1` ）。这种偏置特性通常用一种称为偏置神经元的特殊类型的神经元来表示，它总是输出 1。

图 10-5 表示具有两个输入和三个输出的感知器。该感知器可以将实例同时分类为三个不同的二进制类，这使得它是一个多输出分类器。



那么感知器是如何训练的呢？Frank Rosenblatt 提出的感知器训练算法在很大程度上受到 Hebb 规则的启发。在 1949 出版的《行为组织》一书中，Donald Hebb 提出，当一个生物神经元经常触发另一个神经元时，这两个神经元之间的联系就会变得更强。这个想法后来被 Siegrid Löwel 总结为一个吸引人的短语："一起燃烧的细胞，汇合在一起。"这个规则后来被称为 Hebb 规则（或 HebBIN 学习）；也就是说，当两个神经元具有相同的输出时，它们之间的连接权重就会增加。使用这个规则的变体来训练感知器，该规则考虑了网络所犯的错误；它不加强导致错误输出的连接。更具体地，感知器一次被馈送一个训练实例，并且对于每个实例，它进行预测。对于每一个产生错误预测的输出神经元，它加强了输入的连接权重，这将有助于正确的预测。该规则在公式 10-2 中示出。

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta\left(\hat{y}_j - y_j\right)x_i$$

- 其中 `Wi,j` 是第 `i` 输入神经元与第 `J` 个输出神经元之间的连接权重。
- `xi` 是当前训练实例与输入值。
- `Y` 帽是当前训练实例的第 `J` 个输出神经元的输出。
- `Yj` 是当前训练实例的第 `J` 个输出神经元的目标输出。
- `ε` 是学习率。

每个输出神经元的决策边界是线性的，因此感知机不能学习复杂的模式（就像 Logistic 回归分类器）。然而，如果训练实例是线性可分离的，Rosenblatt 证明该算法将收敛到一个解。这被称为感知器收敛定理。

sklearn 提供了一个感知器类，它实现了一个 LTU 网络。它可以像你所期望的那样使用，例如在 iris 数据集（第 4 章中介绍）：

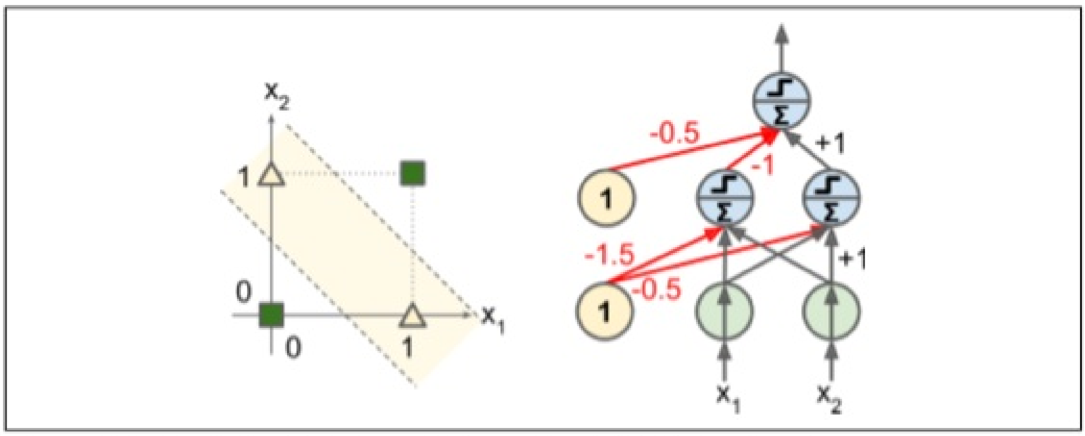
```
import numpy as np
from sklearn.datasets
import load_iris from sklearn.linear_model import Perceptron
iris = load_iris() X = iris.data[:, (2, 3)]  # 花瓣长度，宽度
y = (iris.target == 0).astype(np.int)
per_clf = Perceptron(random_state=42)
per_clf.fit(X, y)
y_pred = per_clf.predict([[2, 0.5]])
```

您可能已经认识到，感知器学习算法类似于随机梯度下降。事实上，sklearn 的感知器类相当于使用具有以下超参数的 SGD 分类器： `loss="perceptron"` ， `learning_rate="constant"` （学习率）， `eta0=1` ， `penalty=None` （无正则化）。

注意，与逻辑斯蒂回归分类器相反，感知机不输出类概率，而是基于硬阈值进行预测。这是你喜欢逻辑斯蒂回归很好的一个理由。

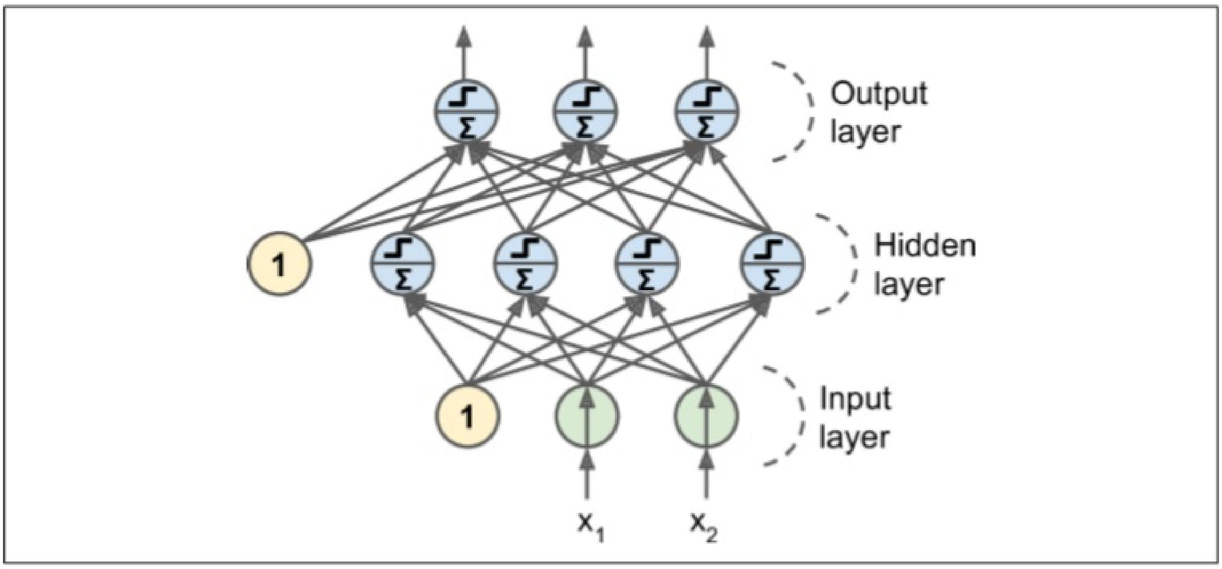
在他们的 1969 个题为“感知者”的专著中，Marvin Minsky 和 Seymour Papert 强调了感知机的许多严重缺陷，特别是它们不能解决一些琐碎的问题（例如，异或（XOR）分类问题）；参见图 10-6 的左侧）。当然，其他的线性分类模型（如 Logistic 回归分类器）也都实现不了，但研究人员期望从感知器中得到更多，他们的失望是很大的：因此，许多研究人员放弃了联结主义（即神经网络的研究），这有利于更高层次的问题，如逻辑、问题解决和搜索。

然而，事实证明，感知器的一些局限性可以通过堆叠多个感知器来消除。由此产生的人工神经网络被称为多层感知器（MLP）。特别地，MLP 可以解决 XOR 问题，因为你可以通过计算图 10-6 右侧所示的 MLP 的输出来验证输入的每一个组合：输入 `(0, 0)` 或 `(1, 1)` 网络输出 0，并且输入 `(0, 1)` 或 `(1, 0)` 它输出 1。



多层感知器与反向传播

MLP 由一个（通过）输入层、一个或多个称为隐藏层的 LTU 组成，一个最终层 LTU 称为输出层（见图 10-7）。除了输出层之外的每一层包括偏置神经元，并且全连接到下一层。当人工神经网络有两个或多个隐含层时，称为深度神经网络（DNN）。



多年来，研究人员努力寻找一种训练 MLP 的方法，但没有成功。但在 1986，D. E. Rumelhart 等人提出了反向传播训练算法。第 9 章我们将其描述为使用反向自动微分的梯度下降（第 4 章讨论了梯度下降，第 9 章讨论了自动微分）。

对于每个训练实例，算法将其馈送到网络并计算每个连续层中的每个神经元的输出（这是向前传递，就像在进行预测时一样）。然后，它测量网络的输出误差（即，期望输出和网络实际输出之间的差值），并且计算最后隐藏层中的每个神经元对每个输出神经元的误差贡献多少。然后，继续测量这些误差贡献有多少来自先前隐藏层中的每个神经元等等，直到算法到达输入层。该反向通过有效地测量网络中所有连接权重的误差梯度，通过在网络中向后传播误差梯度（也是该算法的名称）。如果你查看一下附录 D 中的反向自动微分算法，你会发现反向传播的正向和反向通过简单地执行反向自动微分。反向传播算法的最后一步是使用较早测量的误差梯度对网络中的所有连接权值进行梯度下降步骤。

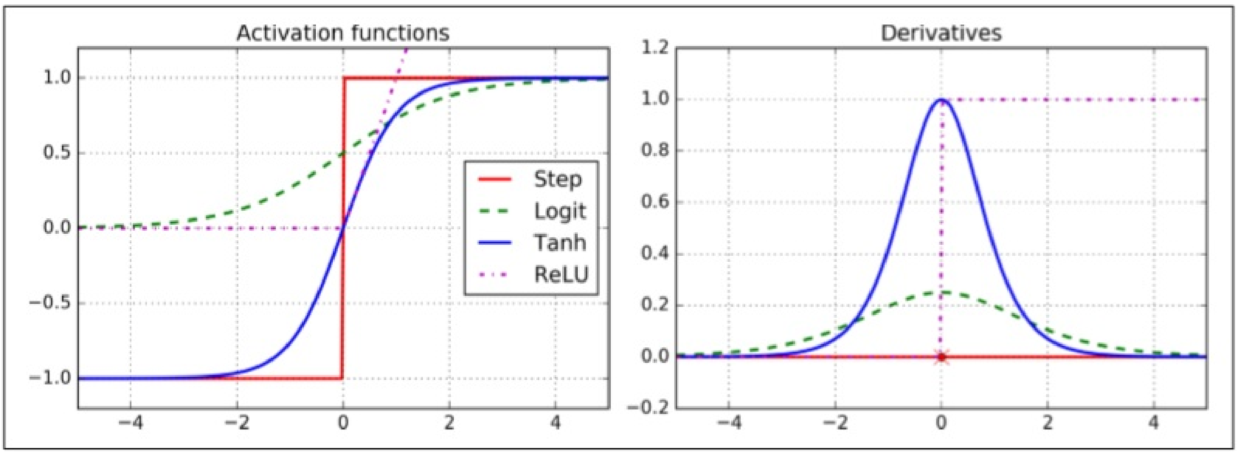
让我们更简短一些：对于每个训练实例，反向传播算法首先进行预测（前向），测量误差，然后反向遍历每个层来测量每个连接（反向传递）的误差贡献，最后稍微调整连接器权值以减少误差（梯度下降步长）。

为了使算法能够正常工作，作者对 MLP 的体系结构进行了一个关键性的改变：用 Logistic 函数代替了阶跃函数，`σ(z) = 1 / (1 + exp(-z))`。这是必要的，因为阶跃函数只包含平坦的段，因此没有梯度来工作（梯度下降不能在平面上移动），而 Logistic 函数到处都有一个定义良好的非零导数，允许梯度下降在每个步上取得一些进展。反向传播算法可以与其他激活函数一起使用，而不是 Logistic 函数。另外两个流行的激活函数是：

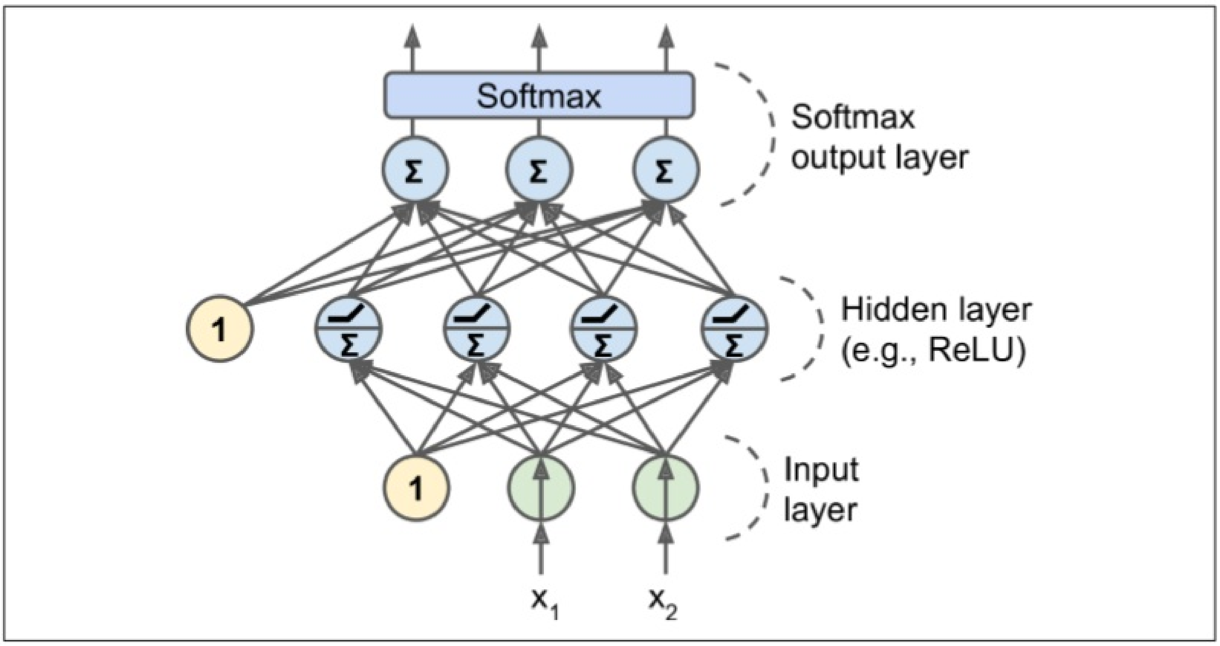
- 双曲正切函数 `tanh (z) = 2σ(2z) - 1`
    - 就像 Logistic 函数，它是 S 形的、连续的、可微的，但是它的输出值范围从-1到 1（不是在 Logistic 函数的 0 到 1），这往往使每个层的输出在训练开始时或多或少都正则化了（即以 0 为中心）。这常常有助于加快收敛速度。
- Relu 函数（在第 9 章中介绍）

- `ReLU (z) = max (0, z)`。它是连续的，但不幸的是在 `z=0` 时不可微（斜率突然改变，这可以使梯度下降反弹）。然而，在实践中，它工作得很好，并且具有快速计算的优点。最重要的是，它没有最大输出值的事实也有助于减少梯度下降期间的一些问题（我们将在第 11 章中回顾这一点）。

这些流行的激活函数及其衍生物如图 10-8 所示。



MLP 通常用于分类，每个输出对应于不同的二进制类（例如，垃圾邮件/正常邮件，紧急/非紧急，等等）。当类是多类的（例如，0 到 9 的数字图像分类）时，输出层通常通过用共享的 softmax 函数替换单独的激活函数来修改（见图 10-9）。第 3 章介绍了 softmax 函数。每个神经元的输出对应于相应类的估计概率。注意，信号只在一个方向上流动（从输入到输出），因此这种结构是前馈神经网络（FNN）的一个例子。



生物神经元似乎是用 sigmoid（S 型）激活函数活动的，因此研究人员在很长一段时间内坚持 sigmoid 函数。但事实证明，Relu 激活函数通常在 ANN 工作得更好。这是生物研究误导的例子之一。

用 TensorFlow 高级 API 训练 MLP

与 TensorFlow 一起训练 MLP 最简单的方法是使用高级 API TF.Learn，这与 sklearn 的 API 非常相似。`DNNClassifier` 可以很容易训练具有任意数量隐层的深度神经网络，而 softmax 输出层输出估计的类概率。例如，下面的代码训练两个隐藏层的 DNN（一个具有 300 个神经元，另一个具有 100 个神经元）和一个具有 10 个神经元的 SOFTMax 输出层进行分类：

```
import tensorflow as tf
feature_columns = tf.contrib.learn.infer_real_valued_columns_from_input(X_train) dnn_c
lf = tf.contrib.learn.DNNClassifier(hidden_units=[300, 100], n_classes=10,
                                    feature_columns=feature_columns)
dnn_clf.fit(x=X_train, y=y_train, batch_size=50, steps=40000)
```

如果你在 MNIST 数据集上运行这个代码（在缩放它之后，例如，通过使用 skLearn 的 `StandardScaler`），你实际上可以得到一个在测试集上达到 98.1% 以上精度的模型！这比我们在第 3 章中训练的最好的模型都要好：

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = list(dnn_clf.predict(X_test))
>>> accuracy_score(y_test, y_pred)
0.98180000000000001
```

TF.Learn 学习库也为评估模型提供了一些方便的功能：

```
>>> dnn_clf.evaluate(X_test, y_test)
{'accuracy': 0.98180002, 'global_step': 40000, 'loss': 0.073678359}
```

`DNNClassifier` 基于 Relu 激活函数创建所有神经元层（我们可以通过设置超参数 `activation_fn` 来改变激活函数）。输出层基于 SoftMax 函数，损失函数是交叉熵（在第 4 章中介绍）。

TF.EXCEL API 仍然是更新的，所以在这些例子中使用的一些名称和函数可能会在你读这本书的时候发生一些变化。但总的思想是不变。

使用普通 TensorFlow 训练 DNN

如果您想要更好地控制网络架构，您可能更喜欢使用 TensorFlow 的较低级别的 Python API（在第 9 章中介绍）。 在本节中，我们将使用与之前的 API 相同的模型，我们将实施 Minibatch 梯度下降来在 MNIST 数据集上进行训练。 第一步是建设阶段，构建 TensorFlow 图。 第二步是执行阶段，您实际运行计算图谱来训练模型。

构造阶段

开始吧。 首先我们需要导入 `tensorflow` 库。 然后我们必须指定输入和输出的数量，并设置每个层中隐藏的神经元数量：

```
import tensorflow as tf
n_inputs = 28*28 # MNIST
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10
```

接下来，与第 9 章一样，您可以使用占位符节点来表示训练数据和目标。 `X` 的形状仅有部分被定义。 我们知道它将是一个 2D 张量（即一个矩阵），沿着第一个维度的实例和第二个维度的特征，我们知道特征的数量将是 `28×28` （每像素一个特征） 但是我们不知道每个训练批次将包含多少个实例。 所以 `X` 的形状是 `(None, n_inputs)` 。 同样，我们知道 `y` 将是一个 1D 张量，每个实例有一个入口，但是我们再次不知道在这一点上训练批次的大小，所以形状 `(None)` 。

```
X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")
```

现在让我们创建一个实际的神经网络。 占位符 `X` 将作为输入层; 在执行阶段，它将一次更换一个训练批次（注意训练批中的所有实例将由神经网络同时处理）。 现在您需要创建两个隐藏层和输出层。 两个隐藏的层几乎相同：它们只是它们所连接的输入和它们包含的神经元的数量不同。 输出层也非常相似，但它使用 softmax 激活函数而不是 ReLU 激活函数。 所以让我们创建一个 `neuron_layer()` 函数，我们将一次创建一个图层。 它将需要参数来指定输入，神经元数量，激活函数和图层的名称：

```
def neuron_layer(X, n_neurons, name, activation=None):
    with tf.name_scope(name):
        n_inputs = int(X.get_shape()[1])
        stddev = 2 / np.sqrt(n_inputs)
        init = tf.truncated_normal((n_inputs, n_neurons), stddev=stddev)
        W = tf.Variable(init, name="weights")
        b = tf.Variable(tf.zeros([n_neurons]), name="biases")
        z = tf.matmul(X, W) + b
        if activation == "relu":
            return tf.nn.relu(z)
        else:
            return z
```

我们逐行浏览这个代码：

1. 首先，我们使用名称范围来创建每层的名称：它将包含该神经元层的所有计算节点。 这是可选的，但如果节点组织良好，则 TensorBoard 图形将会更加出色。

2. 接下来，我们通过查找输入矩阵的形状并获得第二个维度的大小来获得输入数量（第一个维度用于实例）。

3. 接下来的三行创建一个保存权重矩阵的 `W` 变量。 它将是包含每个输入和每个神经元之间的所有连接权重的2D张量；因此，它的形状将是 `(n_inputs, n_neurons)` 。它将被随机初始化，使用具有标准差为 `2/√n` 的截断的正态（高斯）分布(使用截断的正态分布而不是常规正态分布确保不会有任何大的权重，这可能会减慢训练。).使用这个特定的标准差有助于算法的收敛速度更快（我们将在第11章中进一步讨论这一点），这是对神经网络的微小调整之一，对它们的效率产生了巨大的影响）。 重要的是为所有隐藏层随机初始化连接权重，以避免梯度下降算法无法中断的任何对称性。（例如，如果将所有权重设置为 0，则所有神经元将输出 0，并且给定隐藏层中的所有神经元的误差梯度将相同。 然后，梯度下降步骤将在每个层中以相同的方式更新所有权重，因此它们将保持相等。 换句话说，尽管每层有数百个神经元，你的模型就像每层只有一个神经元一样。）
4. 下一行创建一个偏差的 `b` 变量，初始化为 0（在这种情况下无对称问题），每个神经元有一个偏置参数。
5. 然后我们创建一个子图来计算 `z = X·W + b` 。 该向量化实现将有效地计算输入的加权和加上层中每个神经元的偏置，对于批次中的所有实例，仅需一次.
6. 最后，如果激活参数设置为 `relu` ，则代码返回 `relu(z)` （即 `max(0,z)` ），否则它只返回 `z` 。

好了，现在你有一个很好的函数来创建一个神经元层。 让我们用它来创建深层神经网络！ 第一个隐藏层以 `X` 为输入。 第二个将第一个隐藏层的输出作为其输入。 最后，输出层将第二个隐藏层的输出作为其输入。

```
with tf.name_scope("dnn"):
    hidden1 = neuron_layer(X, n_hidden1, "hidden1", activation="relu")
hidden2 = neuron_layer(hidden1, n_hidden2, "hidden2", activation="relu")
logits = neuron_layer(hidden2, n_outputs, "outputs")
```

请注意，为了清楚起见，我们再次使用名称范围。 还要注意，logit 是在通过 softmax 激活函数之前神经网络的输出：为了优化，我们稍后将处理 softmax 计算。

正如你所期望的，TensorFlow 有许多方便的功能来创建标准的神经网络层，所以通常不需要像我们刚才那样定义你自己的 `neuron_layer()` 函数。 例如，TensorFlow 的 `fully_connected()` 函数创建一个完全连接的层，其中所有输入都连接到图层中的所有神经元。 它使用正确的初始化策略来负责创建权重和偏置变量，并且默认情况下使用 ReLU 激活函数（我们可以使用 `activate_fn` 参数来更改它）。 正如我们将在第 11 章中看到的，它还支持正则化和归一化参数。 我们来调整上面的代码来使用 `fully_connected()` 函数，而不是我们的 `neuron_layer()` 函数。 只需导入该功能，并使用以下代码替换 dnn 构建部分：

```
from tensorflow.contrib.layers import fully_connected
with tf.name_scope("dnn"):
    hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
    hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2")
    logits = fully_connected(hidden2, n_outputs, scope="outputs",
                             activation_fn=None)
```

`tensorflow.contrib` 包包含许多有用的功能，但它是一个尚未分级成为主要 TensorFlow API 一部分的实验代码的地方。 因此， `full_connected()` 函数（和任何其他 `contrib` 代码）可能会在将来更改或移动。

使用 `dense()` 代替 `neuron_layer()`

注意：本书使用 `tensorflow.contrib.layers.fully_connected()` 而不是 `tf.layers.dense()` （本章编写时不存在）。

现在最好使用 `tf.layers.dense()` ，因为 `contrib` 模块中的任何内容可能会更改或删除，恕不另行通知。 `dense()` 函数与 `fully_connected()` 函数几乎相同，除了一些细微的差别：

几个参数被重命名： `scope` 变为名称， `activation_fn` 变为激活（同样 `_fn` 后缀从其他参数（如 `normalizer_fn` ）中删除）， `weights_initializer` 成为 `kernel_initializer` 等。默认激活现在是无，而不是 `tf.nn.relu` 。 第 11 章还介绍了更多的差异。

```
with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1",
                              activation=tf.nn.relu)
    hidden2 = tf.layers.dense(hidden1, n_hidden2, name="hidden2",
                              activation=tf.nn.relu)
    logits = tf.layers.dense(hidden2, n_outputs, name="outputs")
```

现在我们已经有了神经网络模型，我们需要定义我们用来训练的损失函数。 正如我们在第 4 章中对 Softmax 回归所做的那样，我们将使用交叉熵。 正如我们之前讨论的，交叉熵将惩罚估计目标类的概率较低的模型。 TensorFlow 提供了几种计算交叉熵的功能。 我们将使用 `sparse_softmax_cross_entropy_with_logits()` ：它根据“logit”计算交叉熵（即，在通过 softmax 激活函数之前的网络输出），并且期望以 0 到 -1 数量的整数形式的标签（在我们的例子中，从 0 到 9）。 这将给我们一个包含每个实例的交叉熵的 1D 张量。 然后，我们可以使用 TensorFlow 的 `reduce_mean()` 函数来计算所有实例的平均交叉熵。

```
with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")
```

该 `sparse_softmax_cross_entropy_with_logits()` 函数等同于应用 SOFTMAX 激活函数，然后计算交叉熵，但它更高效，它妥善照顾的边界情况下，比如 logits 等于 0，这就是为什么我们没有较早的应用 SOFTMAX 激活函数。 还有称为 `softmax_cross_entropy_with_logits()` 的另一个函数，该函数在标签单热载体的形式（而不是整数 0 至类的数目减 1）。

我们有神经网络模型，我们有损失函数，现在我们需要定义一个 `GradientDescentOptimizer` 来调整模型参数以最小化损失函数。没什么新鲜的; 就像我们在第 9 章中所做的那样：

```
learning_rate = 0.01

with tf.name_scope("train"):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    training_op = optimizer.minimize(loss)
```

建模阶段的最后一个重要步骤是指定如何评估模型。 我们将简单地将精度用作我们的绩效指标。 首先，对于每个实例，通过检查最高 logit 是否对应于目标类别来确定神经网络的预测是否正确。 为此，您可以使用 `in_top_k()` 函数。 这返回一个充满布尔值的 1D 张量，因此我们需要将这些布尔值转换为浮点数，然后计算平均值。 这将给我们网络的整体准确性.

```
with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

而且，像往常一样，我们需要创建一个初始化所有变量的节点，我们还将创建一个 `Saver` 来将我们训练有素的模型参数保存到磁盘中：

```
init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

建模阶段结束。 这是不到 40 行代码，但相当激烈：我们为输入和目标创建占位符，我们创建了一个构建神经元层的函数，我们用它来创建 DNN，我们定义了损失函数，我们 创建了一个优化器，最后定义了性能指标。 现在到执行阶段。

执行阶段

这部分要短得多，更简单。 首先，我们加载 MNIST。 我们可以像之前的章节那样使用 ScikitLearn，但是 TensorFlow 提供了自己的助手来获取数据，将其缩放（0 到 1 之间），将它洗牌，并提供一个简单的功能来一次加载一个小批量：

```
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets("/tmp/data/")
```

现在我们定义我们要运行的迭代数，以及小批量的大小：

```
n_epochs = 10001
batch_size = 50
```

现在我们去训练模型:

```python
with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
        acc_test = accuracy.eval(feed_dict={X: mnist.test.images, y: mnist.test.labels
})
        print(epoch, "Train accuracy:", acc_train, "Test accuracy:", acc_test)

    save_path = saver.save(sess, "./my_model_final.ckpt")
```

该代码打开一个 TensorFlow 会话，并运行初始化所有变量的 `init` 节点。 然后它运行的主要训练循环：在每个时期，通过一些小批次的对应于训练集的大小的代码进行迭代。 每个小批量通过 `next_batch()` 方法获取，然后代码简单地运行训练操作，为当前的小批量输入数据和目标提供。 接下来，在每个时期结束时，代码评估最后一个小批量和完整训练集上的模型，并打印出结果。 最后，模型参数保存到磁盘。

使用神经网络

现在神经网络被训练了，你可以用它进行预测。 为此，您可以重复使用相同的建模阶段，但是更改执行阶段，如下所示：

```python
with tf.Session() as sess:
    saver.restore(sess, "./my_model_final.ckpt") # or better, use save_path
    X_new_scaled = mnist.test.images[:20]
    Z = logits.eval(feed_dict={X: X_new_scaled})
    y_pred = np.argmax(Z, axis=1)
```

首先，代码从磁盘加载模型参数。 然后加载一些您想要分类的新图像。 记住应用与训练数据相同的特征缩放（在这种情况下，将其从 0 缩放到 1）。 然后代码评估对数点节点。 如果您想知道所有估计的类概率，则需要将 `softmax()` 函数应用于对数，但如果您只想预测一个类，则可以简单地选择具有最高 logit 值的类（使用 `argmax()` 函数做的伎俩）。

微调神经网络超参数

神经网络的灵活性也是其主要缺点之一：有很多超参数要进行调整。 不仅可以使用任何可想象的网络拓扑（如何神经元互连），而且即使在简单的 MLP 中，您可以更改层数，每层神经元数，每层使用的激活函数类型，权重初始化逻辑等等。 你怎么知道什么组合的超参数是最适合你的任务？

当然，您可以使用具有交叉验证的网格搜索来查找正确的超参数，就像您在前几章中所做的那样，但是由于要调整许多超参数，并且由于在大型数据集上训练神经网络需要很多时间，您只能在合理的时间内探索超参数空间的一小部分。 正如我们在第2章中讨论的那样，使用随

机搜索要好得多。另一个选择是使用诸如 Oscar 之类的工具，它可以实现更复杂的算法，以帮助您快速找到一组好的超参数.

它有助于了解每个超级参数的值是合理的，因此您可以限制搜索空间。 我们从隐藏层数开始。

隐藏层数量

对于许多问题，您只需从单个隐藏层开始，您将获得合理的结果。 实际上已经表明，只有一个隐藏层的 MLP 可以建模甚至最复杂的功能，只要它具有足够的神经元。 长期以来，这些事实说服了研究人员，没有必要调查任何更深层次的神经网络。 但是他们忽略了这样一个事实：深层网络具有比浅层网络更高的参数效率：他们可以使用比浅网格更少的神经元来建模复杂的函数，使得训练更快。

要了解为什么，假设您被要求使用一些绘图软件绘制一个森林，但是您被禁止使用复制/粘贴。 你必须单独绘制每棵树，每枝分枝，每叶叶。 如果你可以画一个叶，复制/粘贴它来绘制一个分支，然后复制/粘贴该分支来创建一个树，最后复制/粘贴这个树来制作一个林，你将很快完成。 现实世界的数据通常以这样一种分层的方式进行结构化，DNN 自动利用这一事实：较低的隐藏层模拟低级结构（例如，各种形状和方向的线段），中间隐藏层将这些低级结构组合到 模型中级结构（例如，正方形，圆形）和最高隐藏层和输出层将这些中间结构组合在一起，以模拟高级结构（如面）。

这种分层架构不仅可以帮助 DNN 更快地融合到一个很好的解决方案，而且还可以提高其将其推广到新数据集的能力。 例如，如果您已经训练了模型以识别图片中的脸部，并且您现在想要训练一个新的神经网络来识别发型，那么您可以通过重新使用第一个网络的较低层次来启动训练。 而不是随机初始化新神经网络的前几层的权重和偏置，您可以将其初始化为第一个网络的较低层的权重和偏置的值。这样，网络将不必从大多数图片中低结构中从头学习；它只需要学习更高层次的结构（例如发型）。

总而言之，对于许多问题，您可以从一个或两个隐藏层开始，它可以正常工作（例如，您可以使用只有一个隐藏层和几百个神经元，在 MNIST 数据集上容易达到 97% 以上的准确度使用两个具有相同总神经元数量的隐藏层，在大致相同的训练时间量中精确度为 98%）。对于更复杂的问题，您可以逐渐增加隐藏层的数量，直到您开始覆盖训练集。非常复杂的任务，例如大型图像分类或语音识别，通常需要具有数十个层（或甚至数百个但不完全相连的网络）的网络，正如我们将在第 13 章中看到的那样），并且需要大量的训练数据。但是，您将很少从头开始训练这样的网络：重用预先训练的最先进的网络执行类似任务的部分更为常见。训练将会更快，需要更少的数据（我们将在第 11 章中进行讨论）

每层隐藏层的神经元数量

显然，输入和输出层中神经元的数量由您的任务需要的输入和输出类型决定。例如，MNIST 任务需要 `28×28 = 784` 个输入神经元和 10 个输出神经元。对于隐藏的层次来说，通常的做法是将其设置为形成一个漏斗，每个层面上的神经元越来越少，原因在于许多低级别功能可以合并成更少的高级功能。例如，MNIST 的典型神经网络可能具有两个隐藏层，第一个具有 300 个神经元，第二个具有 100 个。但是，这种做法现在并不常见，您可以为所有隐藏层使用相同的大小 - 例如，所有隐藏的层与 150 个神经元：这样只用调整一次超参数而不是每层都需要调整（因为如果每层一样，比如 150，之后调就每层都调成 160）。就像层数一样，您可以尝试逐渐增加神经元的数量，直到网络开始过度拟合。一般来说，通过增加每层的神经元数量，可以增加层数，从而获得更多的消耗。不幸的是，正如你所看到的，找到完美的神经元数量仍然是黑色的艺术.

一个更简单的方法是选择一个具有比实际需要的更多层次和神经元的模型，然后使用早期停止来防止它过度拟合（以及其他正则化技术，特别是 drop out，我们将在第 11 章中看到）。这被称为“拉伸裤”的方法：而不是浪费时间寻找完美匹配您的大小的裤子，只需使用大型伸缩裤，缩小到合适的尺寸。

激活函数

在大多数情况下，您可以在隐藏层中使用 ReLU 激活函数（或其中一个变体，我们将在第 11 章中看到）。 与其他激活函数相比，计算速度要快一些，而梯度下降在局部最高点上并不会被卡住，因为它不会对大的输入值饱和（与逻辑函数或双曲正切函数相反, 他们容易在 1 饱和)

对于输出层，softmax 激活函数通常是分类任务的良好选择（当这些类是互斥的时）。 对于回归任务，您完全可以不使用激活函数。

这就是人造神经网络的这个介绍。 在接下来的章节中，我们将讨论训练非常深的网络的技术，并分发多个服务器和 GPU 的训练。 然后我们将探讨一些其他流行的神经网络架构：卷积神经网络，循环神经网络和自动编码器。

完整代码

```
from tensorflow.examples.tutorials.mnist import input_data
import tensorflow as tf
from sklearn.metrics import accuracy_score
import numpy as np

if __name__ == '__main__':
    n_inputs = 28 * 28
    n_hidden1 = 300
    n_hidden2 = 100
    n_outputs = 10

    mnist = input_data.read_data_sets("/tmp/data/")

    X_train = mnist.train.images
    X_test = mnist.test.images
    y_train = mnist.train.labels.astype("int")
    y_test = mnist.test.labels.astype("int")

    X = tf.placeholder(tf.float32, shape= (None, n_inputs), name='X')
    y = tf.placeholder(tf.int64, shape=(None), name = 'y')

    with tf.name_scope('dnn'):
        hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu
                                  ,name= 'hidden1')

        hidden2 = tf.layers.dense(hidden1, n_hidden2, name='hidden2',
                                  activation= tf.nn.relu)

        logits = tf.layers.dense(hidden2, n_outputs, name='outputs')

    with tf.name_scope('loss'):
        xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels = y,
                                                                  logits = logits)
        loss = tf.reduce_mean(xentropy, name='loss')#所有值求平均

    learning_rate = 0.01

    with tf.name_scope('train'):
        optimizer = tf.train.GradientDescentOptimizer(learning_rate)
        training_op = optimizer.minimize(loss)

    with tf.name_scope('eval'):
        correct = tf.nn.in_top_k(logits ,y ,1)#是否与真值一致 返回布尔值
        accuracy = tf.reduce_mean(tf.cast(correct, tf.float32)) #tf.cast将数据转化为0,1
序列

    init = tf.global_variables_initializer()

    n_epochs = 20
    batch_size = 50
    with tf.Session() as sess:
        init.run()
        for epoch in range(n_epochs):
            for iteration in range(mnist.train.num_examples // batch_size):
                X_batch, y_batch = mnist.train.next_batch(batch_size)
                sess.run(training_op,feed_dict={X:X_batch,
                                                y: y_batch})
            acc_train = accuracy.eval(feed_dict={X:X_batch,
                                                y: y_batch})
            acc_test = accuracy.eval(feed_dict={X: mnist.test.images,
                                                y: mnist.test.labels})
            print(epoch, "Train accuracy:", acc_train, "Test accuracy:", acc_test)
```

练习

1. 使用原始的人工神经元（如图 10-3 中的一个）来计算神经网络，计算`A ⊕ B`（`⊕`表示 XOR 运算）。提示：`A ⊕ B = (A ∧ ¬ B) ∨ (¬ A ∧ B)`。
2. 为什么通常使用逻辑斯蒂回归分类器而不是经典感知器（即使用感知器训练算法训练单层的线性阈值单元）？你如何调整感知器使之等同于逻辑回归分类器？
3. 为什么激活函数是训练第一个 MLP 的关键因素？
4. 说出三种流行的激活函数。你能画出它们吗？
5. 假设有一个 MLP 有一个 10 个神经元组成的输入层，接着是一个 50 个神经元的隐藏层，最后一个 3 个神经元输出层。所有人工神经元使用 Relu 激活函数。
    - 输入矩阵`X`的形状是什么？
    - 隐藏层的权重向量的形状以及它的偏置向量的形状如何？
    - 输出层的权重向量和它的偏置向量的形状是什么？
    - 网络的输出矩阵`Y`是什么形状？
    - 写出计算网络输出矩阵的方程
6. 如果你想把电子邮件分类成垃圾邮件或正常邮件，你需要在输出层中有多少个神经元？在输出层中应该使用什么样的激活函数？如果你想解决 MNIST 问题，你需要多少神经元在输出层，使用什么激活函数？如第 2 章，一样让你的网络预测房屋价格。
7. 什么是反向传播，它是如何工作的？反向传播与反向自动微分有什么区别？
8. 你能列出所有可以在 MLP 中调整的超参数吗？如果 MLP 与训练数据相匹配，你如何调整这些超参数来解决这个问题？
9. 在 MNIST 数据集上训练一个深层 MLP 并查看是否可以超过 98% 的精度。就像在第 9 章的最后一次练习中，尝试添加所有的铃声和哨子（即，保存检查点，在中断的情况下恢复最后一个检查点，添加摘要，使用 TensorBoard 绘制学习曲线，等等）。

练习的答案请参照附录 A

第11章 训练深层神经网络

第 10 章介绍了人工神经网络，并训练了我们的第一个深度神经网络。 但它是一个非常浅的 DNN，只有两个隐藏层。 如果你需要解决非常复杂的问题，例如检测高分辨率图像中的数百种类型的对象，该怎么办？ 你可能需要训练更深的 DNN，也许有 10 层，每层包含数百个神经元，通过数十万个连接来连接。 这不会是闲庭信步：

- 首先，你将面临棘手的梯度消失问题（或相关的梯度爆炸问题），这会影响深度神经网络，并使较低层难以训练。
- 其次，对于如此庞大的网络，训练将非常缓慢。
- 第三，具有数百万参数的模型将会有严重的过拟合训练集的风险。

在本章中，我们将依次讨论这些问题，并提出解决问题的技巧。 我们将从解释梯度消失问题开始，并探讨解决这个问题的一些最流行的解决方案。 接下来我们将看看各种优化器，与普通梯度下降相比，它们可以加速大型模型的训练。 最后，我们将浏览一些流行的大型神经网络正则化技术。

使用这些工具，你将能够训练非常深的网络：欢迎来到深度学习的世界！

梯度消失/爆炸问题

正如我们在第 10 章中所讨论的那样，反向传播算法的工作原理是从输出层到输入层，传播误差的梯度。 一旦该算法已经计算了网络中每个参数的损失函数的梯度，它就使用这些梯度来用梯度下降步骤来更新每个参数。

不幸的是，梯度往往变得越来越小，随着算法进展到较低层。 结果，梯度下降更新使得低层连接权重实际上保持不变，并且训练永远不会收敛到良好的解决方案。 这被称为梯度消失问题。 在某些情况下，可能会发生相反的情况：梯度可能变得越来越大，许多层得到了非常大的权重更新，算法发散。这是梯度爆炸的问题，在循环神经网络中最为常见（见第 14 章）。更一般地说，深度神经网络受梯度不稳定之苦; 不同的层次可能以非常不同的速度学习。

虽然这种不幸的行为已经经过了相当长的一段时间的实验观察（这是造成深度神经网络大部分时间都被抛弃的原因之一），但直到 2010 年左右，人们才有了明显的进步。 Xavier Glorot 和 Yoshua Bengio 发表的题为《Understanding the Difficulty of Training Deep Feedforward Neural Networks》的论文发现了一些疑问，包括流行的 sigmoid 激活函数和当时最受欢迎的权重初始化技术的组合，即随机初始化时使用平均值为 0，标准差为 1 的正态分布。简而言之，他们表明，用这个激活函数和这个初始化方案，每层输出的方差远大于其输入的方差。网络正向，每层的方差持续增加，直到激活函数在顶层饱和。这实际上是因为logistic函数的平均值为 0.5 而不是 0（双曲正切函数的平均值为 0，表现略好于深层网络中的logistic函数）

看一下logistic 激活函数（参见图 11-1），可以看到当输入变大（负或正）时，函数饱和在 0 或 1，导数非常接近 0。因此，当反向传播开始时， 它几乎没有梯度通过网络传播回来，而且由于反向传播通过顶层向下传递，所以存在的小梯度不断地被稀释，因此较低层确实没有任何东西可用。

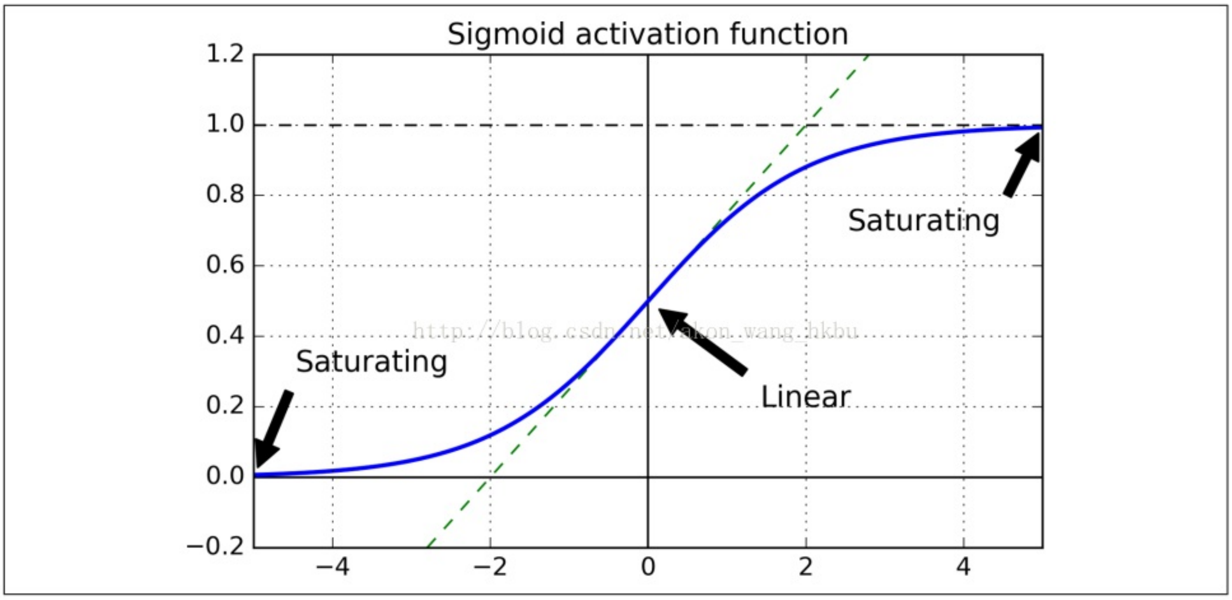


Figure 11-1. Logistic activation function saturation

Glorot 和 Bengio 在他们的论文中提出了一种显著缓解这个问题的方法。 我们需要信号在两个方向上正确地流动：在进行预测时是正向的，在反向传播梯度时是反向的。 我们不希望信号消失，也不希望它爆炸并饱和。 为了使信号正确流动，作者认为，我们需要每层输出的方差等于其输入的方差。（这里有一个比喻：如果将麦克风放大器的旋钮设置得太接近于零，人们听不到声音，但是如果将麦克风放大器设置得太大，声音就会饱和，人们就会听不懂你在说什么。 现在想象一下这样一个放大器的链条：它们都需要正确设置，以便在链条的末端响亮而清晰地发出声音。 你的声音必须以每个放大器的振幅相同的幅度出来。）而且我们也需要梯度在相反方向上流过一层之前和之后有相同的方差（如果您对数学细节感兴趣，请查阅论文）。实际上不可能保证两者都是一样的，除非这个层具有相同数量的输入和输出连接，但是他们提出了一个很好的折衷办法，在实践中证明这个折中办法非常好：随机初始化连接权重必须如公式 11-1 所描述的那样。其中 `n_inputs` 和 `n_outputs` 是权重正在被初始化的层（也称为扇入和扇出）的输入和输出连接的数量。 这种初始化策略通常被称为Xavier初始化（在作者的名字之后），或者有时是 Glorot 初始化。

Equation 11-1. Xavier initialization (when using the logistic activation function)

$$\text{Normal distribution with mean 0 and standard deviation } \sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$$

［数］均匀分布

$$\text{Or a uniform distribution between -r and +r, with } r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$$

当输入连接的数量大致等于输出连接的数量时，可以得到更简单的等式 (e.g., $\sigma = 1/\sqrt{n_{\text{inputs}}}$ or $r = \sqrt{3}/\sqrt{n_{\text{inputs}}}$). 我们在第 10 章中使用了这个简化的策略。

使用 Xavier 初始化策略可以大大加快训练速度，这是导致深度学习目前取得成功的技巧之一。 最近的一些论文针对不同的激活函数提供了类似的策略，如表 11-1 所示。 ReLU 激活函数（及其变体，包括简称 ELU 激活）的初始化策略有时称为 He 初始化（在其作者的姓氏之后）。

Table 11-1. Initialization parameters for each type of activation function

Activation function	Uniform distribution [–r, r]	Normal distribution
Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

默认情况下， `fully_connected()` 函数（在第 10 章中介绍）使用 Xavier 初始化（具有统一的分布）。 你可以通过使用如下所示的 `variance_scaling_initializer()` 函数来将其更改为 He 初始化：

注意：本书使用 `tensorflow.contrib.layers.fully_connected()` 而不是 `tf.layers.dense()` （本章编写时不存在）。 现在最好使用 `tf.layers.dense()` ，因为 `contrib` 模块中的任何内容可能会更改或删除，恕不另行通知。 `dense()` 函数几乎与 `fully_connected()` 函数完全相同。 与本章有关的主要差异是：

几个参数被重新命名：范围变成名字， `activation_fn` 变成激活（类似地， `_fn` 后缀从诸如 `normalizer_fn` 之类的其他参数中移除）， `weights_initializer` 变成 `kernel_initializer` 等等。默认激活现在是 `None` ，而不是 `tf.nn.relu` 。 它不支持 `tensorflow.contrib.framework.arg_scope()` （稍后在第 11 章中介绍）。 它不支持正则化的参数（稍后在第 11 章介绍）。

```
he_init = tf.contrib.layers.variance_scaling_initializer()
hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu,
                          kernel_initializer=he_init, name="hidden1")
```

He 初始化只考虑了扇入，而不是像 Xavier 初始化那样扇入和扇出之间的平均值。 这也是 `variance_scaling_initializer()` 函数的默认值，但您可以通过设置参数 `mode ="FAN_AVG"` 来更改它。

非饱和激活函数

Glorot 和 Bengio 在 2010 年的论文中的一个见解是，消失/爆炸的梯度问题部分是由于激活函数的选择不好造成的。 在那之前，大多数人都认为，如果大自然选择在生物神经元中使用 sigmoid 激活函数，它们必定是一个很好的选择。 但事实证明，其他激活函数在深度神经网络中表现得更好，特别是 ReLU 激活函数，主要是因为它对正值不会饱和（也因为它的计算速度很快）。

不幸的是，ReLU激活功能并不完美。 它有一个被称为 “ReLU 死区” 的问题：在训练过程中，一些神经元有效地死亡，意味着它们停止输出 0 以外的任何东西。在某些情况下，你可能会发现你网络的一半神经元已经死亡，特别是如果你使用大学习率。 在训练期间，如果神经元的权重得到更新，使得神经元输入的加权和为负，则它将开始输出 0 。当这种情况发生时，由于当输入为负时，ReLU函数的梯度为0，神经元不可能恢复生机。

为了解决这个问题，你可能需要使用 ReLU 函数的一个变体，比如 leaky ReLU。这个函数定义为 `LeakyReLUα(z)= max(αz，z)` （见图 11-2）。超参数 `α` 定义了函数“leaks”的程度：它是 `z < 0` 时函数的斜率，通常设置为 0.01。这个小斜坡确保 leaky ReLU 永不死亡；他们可能会长期昏迷，但他们有机会最终醒来。最近的一篇论文比较了几种 ReLU 激活功能的变体，其中一个结论是 leaky Relu 总是优于严格的 ReLU 激活函数。事实上，设定 `α= 0.2` （巨大 leak）似乎导致比 `α= 0.01` （小 leak）更好的性能。他们还评估了随机化 leaky ReLU（RReLU），其中 `α` 在训练期间在给定范围内随机挑选，并在测试期间固定为平均值。它表现相当好，似乎是一个正则项（减少训练集的过拟合风险）。最后，他们还评估了参数 leaky ReLU（PReLU），其中 `α` 被授权在训练期间被学习（而不是超参数，它变成可以像任何其他参数一样被反向传播修改的参数）。据报道这在大型图像数据集上的表现强于 ReLU，但是对于较小的数据集，其具有过度拟合训练集的风险。

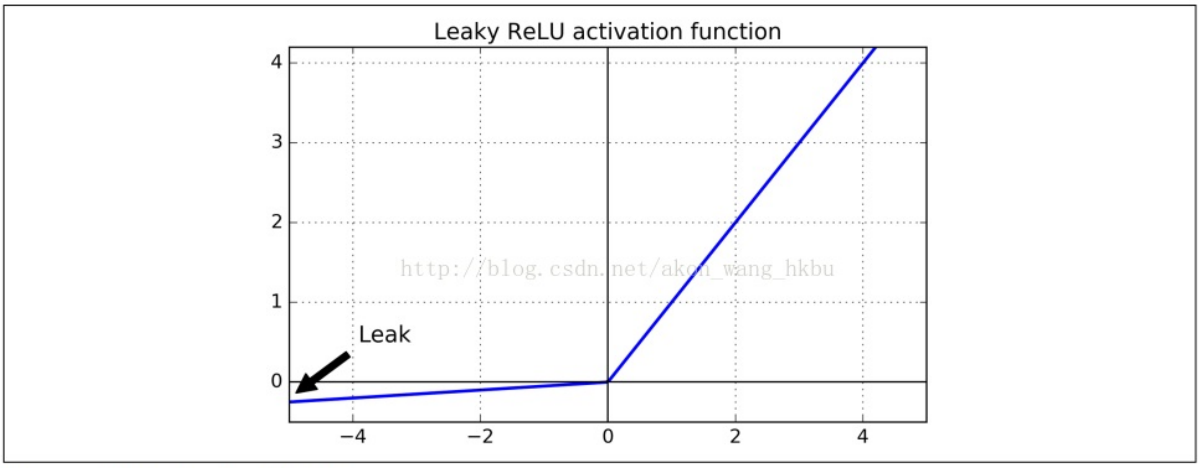


Figure 11-2. Leaky ReLU

最后，Djork-Arné Clevert 等人在 2015 年的一篇论文中提出了一种称为指数线性单元（exponential linear unit，ELU）的新的激活函数，在他们的实验中表现优于所有的 ReLU 变体：训练时间减少，神经网络在测试集上表现的更好。 如图 11-3 所示，公式 11-2 给出了它的定义。

Equation 11-2. ELU activation function

$$\mathrm{ELU}_{\alpha}(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & if z \geq 0 \end{cases}$$

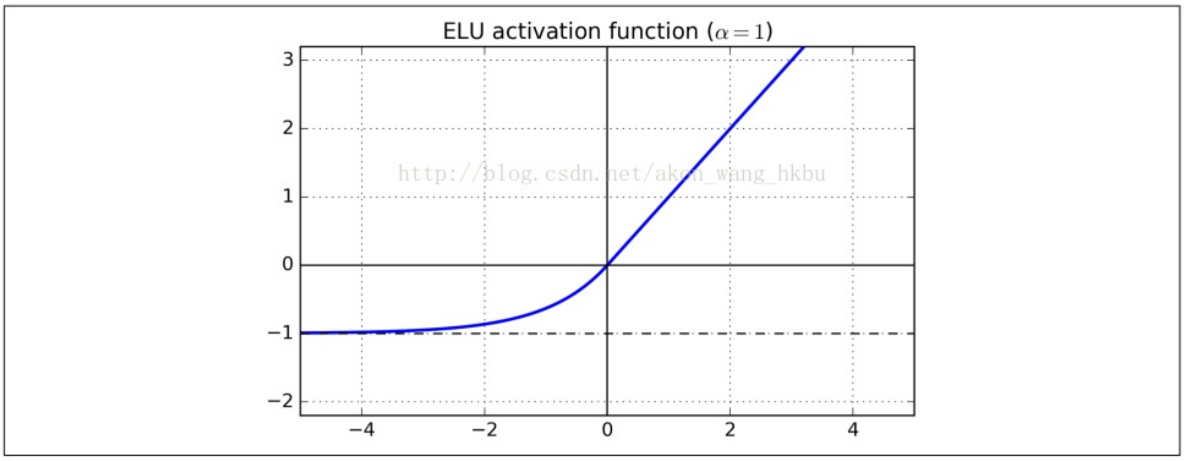


Figure 11-3. ELU activation function

它看起来很像 ReLU 函数，但有一些区别，主要区别在于：

- 首先它在 `z < 0` 时取负值，这使得该单元的平均输出接近于 0。这有助于减轻梯度消失问题，如前所述。 超参数 `α` 定义为当 `z` 是一个大的负数时，ELU 函数接近的值。它通常设置为 1，但是如果你愿意，你可以像调整其他超参数一样调整它。
- 其次，它对 `z < 0` 有一个非零的梯度，避免了神经元死亡的问题。
- 第三，函数在任何地方都是平滑的，包括 `z = 0` 左右，这有助于加速梯度下降，因为它不会弹回 `z = 0` 的左侧和右侧。

ELU 激活函数的主要缺点是计算速度慢于 ReLU 及其变体（由于使用指数函数），但是在训练过程中，这是通过更快的收敛速度来补偿的。 然而，在测试时间，ELU 网络将比 ReLU 网络慢。

那么你应该使用哪个激活函数来处理深层神经网络的隐藏层？ 虽然你的里程会有所不同，一般 ELU > leaky ReLU（及其变体）> ReLU > tanh > sigmoid。 如果您关心运行时性能，那么您可能喜欢 leaky ReLU超过ELU。 如果你不想调整另一个超参数，你可以使用前面提到的默认的 `α` 值（leaky ReLU 为 0.01，ELU 为 1）。 如果您有充足的时间和计算能力，您可以使用交叉验证来评估其他激活函数，特别是如果您的神经网络过拟合，则为RReLU; 如果您拥有庞大的训练数据集，则为 PReLU。

TensorFlow 提供了一个可以用来建立神经网络的 `elu()` 函数。 调用 `fully_connected()` 函数时，只需设置 `activation_fn` 参数即可：

```
hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.elu, name="hidden1")
```

TensorFlow 没有针对 leaky ReLU 的预定义函数，但是很容易定义：

```
def leaky_relu(z, name=None):
    return tf.maximum(0.01 * z, z, name=name)

hidden1 = tf.layers.dense(X, n_hidden1, activation=leaky_relu, name="hidden1")
```

批量标准化

尽管使用 He初始化和 ELU（或任何 ReLU 变体）可以显著减少训练开始阶段的梯度消失/爆炸问题，但不保证在训练期间问题不会回来。

在 2015 年的一篇论文中，Sergey Ioffe 和 Christian Szegedy 提出了一种称为批量标准化（Batch Normalization，BN）的技术来解决梯度消失/爆炸问题，每层输入的分布在训练期间改变的问题，更普遍的问题是当前一层的参数改变，每层输入的分布会在训练过程中发生变化（他们称之为内部协变量偏移问题）。

该技术包括在每层的激活函数之前在模型中添加操作，简单地对输入进行zero-centering和规范化，然后每层使用两个新参数（一个用于尺度变换，另一个用于偏移）对结果进行尺度变换和偏移。 换句话说，这个操作可以让模型学习到每层输入值的最佳尺度,均值。为了对输入进行归零和归一化，算法需要估计输入的均值和标准差。 它通过评估当前小批量输入的均值和标准差（因此命名为“批量标准化”）来实现。 整个操作在方程 11-3 中。

Equation 11-3. Batch Normalization algorithm

1. $$\mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$$

2. $$\sigma_B{}^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} \left(\mathbf{x}^{(i)} - \mu_B\right)^2$$

3. $$\widehat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B{}^2 + \epsilon}}$$

4. $$\mathbf{z}^{(i)} = \gamma \widehat{\mathbf{x}}^{(i)} + \beta$$

μ_B 是整个小批量B的经验均值

σ_B 是经验性的标准差，也是来评估整个小批量的。

m_B 是小批量中的实例数量。

$\hat{x}^{(i)}$ 是以为零中心和标准化的输入。

γ 是层的缩放参数。

β 是层的移动参数（偏移量）

ϵ 是一个很小的数字，以避免被零除（通常为 `10 ^ -3`）。 这被称为平滑项（拉布拉斯平滑，Laplace Smoothing）。

$z^{(i)}$ 是BN操作的输出：它是输入的缩放和移位版本。

在测试时，没有小批量计算经验均值和标准差，所以您只需使用整个训练集的均值和标准差。 这些通常在训练期间使用移动平均值进行有效计算。 因此，总的来说，每个批次标准化的层次都学习了四个参数：`γ`（标度），`β`（偏移），`μ`（平均值）和 `σ`（标准差）。

作者证明，这项技术大大改善了他们试验的所有深度神经网络。梯度消失问题大大减少了，他们可以使用饱和激活函数，如 tanh 甚至 sigmoid 激活函数。网络对权重初始化也不那么敏感。他们能够使用更大的学习率，显著加快了学习过程。具体地，他们指出，“应用于最先进的图像分类模型，批标准化用少了 14 倍的训练步骤实现了相同的精度，以显著的优势击败了原始模型。[...] 使用批量标准化的网络集合，我们改进了 ImageNet 分类上的最佳公布结果：达到4.9% 的前5个验证错误（和 4.8% 的测试错误），超出了人类评估者的准确性。批量标准化也像一个正则化项一样，减少了对其他正则化技术的需求（如本章稍后描述的 dropout）.

然而，批量标准化的确会增加模型的复杂性（尽管它不需要对输入数据进行标准化，因为第一个隐藏层会照顾到这一点，只要它是批量标准化的）。 此外，还存在运行时间的损失：由于每层所需的额外计算，神经网络的预测速度较慢。 所以，如果你需要预测闪电般快速，你可能想要检查普通ELU + He初始化执行之前如何执行批量标准化。

您可能会发现，训练起初相当缓慢，而渐变下降正在寻找每层的最佳尺度和偏移量，但一旦找到合理的好值，它就会加速。

使用 TensorFlow 实现批量标准化

TensorFlow 提供了一个 `batch_normalization()` 函数，它简单地对输入进行居中和标准化，但是您必须自己计算平均值和标准差（基于训练期间的小批量数据或测试过程中的完整数据集） 作为这个函数的参数，并且还必须处理缩放和偏移量参数的创建（并将它们传递给此函数）。 这是可行的，但不是最方便的方法。 相反，你应该使用 `batch_norm()` 函数，它为你处理所有这些。 您可以直接调用它，或者告诉 `fully_connected()` 函数使用它，如下面的代码所示：

注意：本书使用 `tensorflow.contrib.layers.batch_norm()` 而不是 `tf.layers.batch_normalization()`（本章写作时不存在）。 现在最好使用 `tf.layers.batch_normalization()`，因为 `contrib` 模块中的任何内容都可能会改变或被删除，恕不另行通知。 我们现在不使用 `batch_norm()` 函数作为 `fully_connected()` 函数的正则化参数，而是使用 `batch_normalization()`，并明确地创建一个不同的层。 参数有些不同，特别是：

- `decay` 更名为 `momentum`
- `is_training` 被重命名为 `training`
- `updates_collections` 被删除：批量标准化所需的更新操作被添加到 `UPDATE_OPS` 集合中，并且您需要在训练期间明确地运行这些操作（请参阅下面的执行阶段）
- 我们不需要指定 `scale = True`，因为这是默认值。

还要注意，为了在每个隐藏层激活函数之前运行批量标准化，我们手动应用 RELU 激活函数，在批量规范层之后。注意：由于 `tf.layers.dense()` 函数与本书中使用的 `tf.contrib.layers.arg_scope()` 不兼容，我们现在使用 python 的 `functools.partial()` 函数。 它可以很容易地创建一个 `my_dense_layer()` 函数，只需调用 `tf.layers.dense()`，并自动设置所需的参数（除非在调用 `my_dense_layer()` 时覆盖它们）。 如您所见，代码保持非常相似。

```python
import tensorflow as tf

n_inputs = 28 * 28
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")

training = tf.placeholder_with_default(False, shape=(), name='training')

hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1")
bn1 = tf.layers.batch_normalization(hidden1, training=training, momentum=0.9)
bn1_act = tf.nn.elu(bn1)

hidden2 = tf.layers.dense(bn1_act, n_hidden2, name="hidden2")
bn2 = tf.layers.batch_normalization(hidden2, training=training, momentum=0.9)
bn2_act = tf.nn.elu(bn2)

logits_before_bn = tf.layers.dense(bn2_act, n_outputs, name="outputs")
logits = tf.layers.batch_normalization(logits_before_bn, training=training,
                                       momentum=0.9)
```

```python
X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
training = tf.placeholder_with_default(False, shape=(), name='training')
```

为了避免一遍又一遍重复相同的参数，我们可以使用 Python 的 `partial()` 函数：

```
from functools import partial

my_batch_norm_layer = partial(tf.layers.batch_normalization,
                              training=training, momentum=0.9)

hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1")
bn1 = my_batch_norm_layer(hidden1)
bn1_act = tf.nn.elu(bn1)
hidden2 = tf.layers.dense(bn1_act, n_hidden2, name="hidden2")
bn2 = my_batch_norm_layer(hidden2)
bn2_act = tf.nn.elu(bn2)
logits_before_bn = tf.layers.dense(bn2_act, n_outputs, name="outputs")
logits = my_batch_norm_layer(logits_before_bn)
```

完整代码

```
from functools import partial
from tensorflow.examples.tutorials.mnist import input_data
import tensorflow as tf

if __name__ == '__main__':
    n_inputs = 28 * 28
    n_hidden1 = 300
    n_hidden2 = 100
    n_outputs = 10

    mnist = input_data.read_data_sets("/tmp/data/")

    batch_norm_momentum = 0.9
    learning_rate = 0.01

    X = tf.placeholder(tf.float32, shape=(None, n_inputs), name = 'X')
    y = tf.placeholder(tf.int64, shape=None, name = 'y')
    training = tf.placeholder_with_default(False, shape=(), name = 'training')#给Batch
norm加一个placeholder

    with tf.name_scope("dnn"):
        he_init = tf.contrib.layers.variance_scaling_initializer()
        #对权重的初始化

        my_batch_norm_layer = partial(
            tf.layers.batch_normalization,
            training = training,
            momentum = batch_norm_momentum
        )

        my_dense_layer = partial(
            tf.layers.dense,
            kernel_initializer = he_init
        )

        hidden1 = my_dense_layer(X ,n_hidden1 ,name = 'hidden1')
        bn1 = tf.nn.elu(my_batch_norm_layer(hidden1))
        hidden2 = my_dense_layer(bn1, n_hidden2, name = 'hidden2')
        bn2 = tf.nn.elu(my_batch_norm_layer(hidden2))
        logists_before_bn = my_dense_layer(bn2, n_outputs, name = 'outputs')
        logists = my_batch_norm_layer(logists_before_bn)

    with tf.name_scope('loss'):
        xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels = y, logits=
logists)
        loss = tf.reduce_mean(xentropy, name = 'loss')

    with tf.name_scope('train'):
        optimizer = tf.train.GradientDescentOptimizer(learning_rate)
        training_op = optimizer.minimize(loss)

    with tf.name_scope("eval"):
```

```
        correct = tf.nn.in_top_k(logists, y, 1)
        accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

    init = tf.global_variables_initializer()
    saver = tf.train.Saver()

    n_epoches = 20
    batch_size = 200
# 注意：由于我们使用的是 tf.layers.batch_normalization() 而不是 tf.contrib.layers.batch_norm()（如本书所述），
# 所以我们需要明确运行批量规范化所需的额外更新操作（sess.run([ training_op，extra_update_ops], ...)。
    extra_update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)

    with tf.Session() as sess:
        init.run()
        for epoch in range(n_epoches):
            for iteraton in range(mnist.train.num_examples//batch_size):
                X_batch, y_batch = mnist.train.next_batch(batch_size)
                sess.run([training_op,extra_update_ops],
                         feed_dict={training:True, X:X_batch, y:y_batch})
            accuracy_val = accuracy.eval(feed_dict= {X:mnist.test.images,
                                                    y:mnist.test.labels})
            print(epoch, 'Test accuracy:', accuracy_val)
```

```
0 Test accuracy: 0.865
1 Test accuracy: 0.8951
2 Test accuracy: 0.9097
3 Test accuracy: 0.919
4 Test accuracy: 0.9279
5 Test accuracy: 0.9344
6 Test accuracy: 0.9384
7 Test accuracy: 0.9436
8 Test accuracy: 0.9473
9 Test accuracy: 0.9515
10 Test accuracy: 0.9536
11 Test accuracy: 0.9539
12 Test accuracy: 0.9564
13 Test accuracy: 0.9593
14 Test accuracy: 0.9598
15 Test accuracy: 0.9611
16 Test accuracy: 0.9631
17 Test accuracy: 0.964
18 Test accuracy: 0.9649
19 Test accuracy: 0.9652
```

什么！？ 这对 MNIST 来说不是一个很好的准确性。 当然，如果你训练的时间越长，准确性就越好，但是由于这样一个浅的网络，批量范数和 ELU 不太可能产生非常积极的影响：它们大部分都是为了更深的网络而发光。请注意，您还可以训练操作取决于更新操作：

```
with tf.name_scope("train"):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    extra_update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
    with tf.control_dependencies(extra_update_ops):
        training_op = optimizer.minimize(loss)
```

这样，你只需要在训练过程中评估training_op，TensorFlow也会自动运行更新操作：

```
sess.run(training_op, feed_dict={training: True, X: X_batch, y: y_batch})
```

梯度裁剪

减少梯度爆炸问题的一种常用技术是在反向传播过程中简单地剪切梯度，使它们不超过某个阈值（这对于递归神经网络是非常有用的；参见第 14 章）。 这就是所谓的梯度裁剪。一般来说，人们更喜欢批量标准化，但了解梯度裁剪以及如何实现它仍然是有用的。

在 TensorFlow 中，优化器的 `minimize()` 函数负责计算梯度并应用它们，所以您必须首先调用优化器的 `compute_gradients()` 方法，然后使用 `clip_by_value()` 函数创建一个裁剪梯度的操作，最后 创建一个操作来使用优化器的 `apply_gradients()` 方法应用裁剪梯度：

```
threshold = 1.0

optimizer = tf.train.GradientDescentOptimizer(learning_rate)
grads_and_vars = optimizer.compute_gradients(loss)
capped_gvs = [(tf.clip_by_value(grad, -threshold, threshold), var)
              for grad, var in grads_and_vars]
training_op = optimizer.apply_gradients(capped_gvs)
```

像往常一样，您将在每个训练阶段运行这个 `training_op`。 它将计算梯度，将它们裁剪到 -1.0 和 1.0 之间，并应用它们。 `threhold` 是您可以调整的超参数。

复用预训练层

从零开始训练一个非常大的 DNN 通常不是一个好主意，相反，您应该总是尝试找到一个现有的神经网络来完成与您正在尝试解决的任务类似的任务，然后复用这个网络的较低层：这就是所谓的迁移学习。这不仅会大大加快训练速度，还将需要更少的训练数据。

例如，假设您可以访问经过训练的 DNN，将图片分为 100 个不同的类别，包括动物，植物，车辆和日常物品。 您现在想要训练一个 DNN 来对特定类型的车辆进行分类。 这些任务非常相似，因此您应该尝试重新使用第一个网络的一部分（请参见图 11-4）。

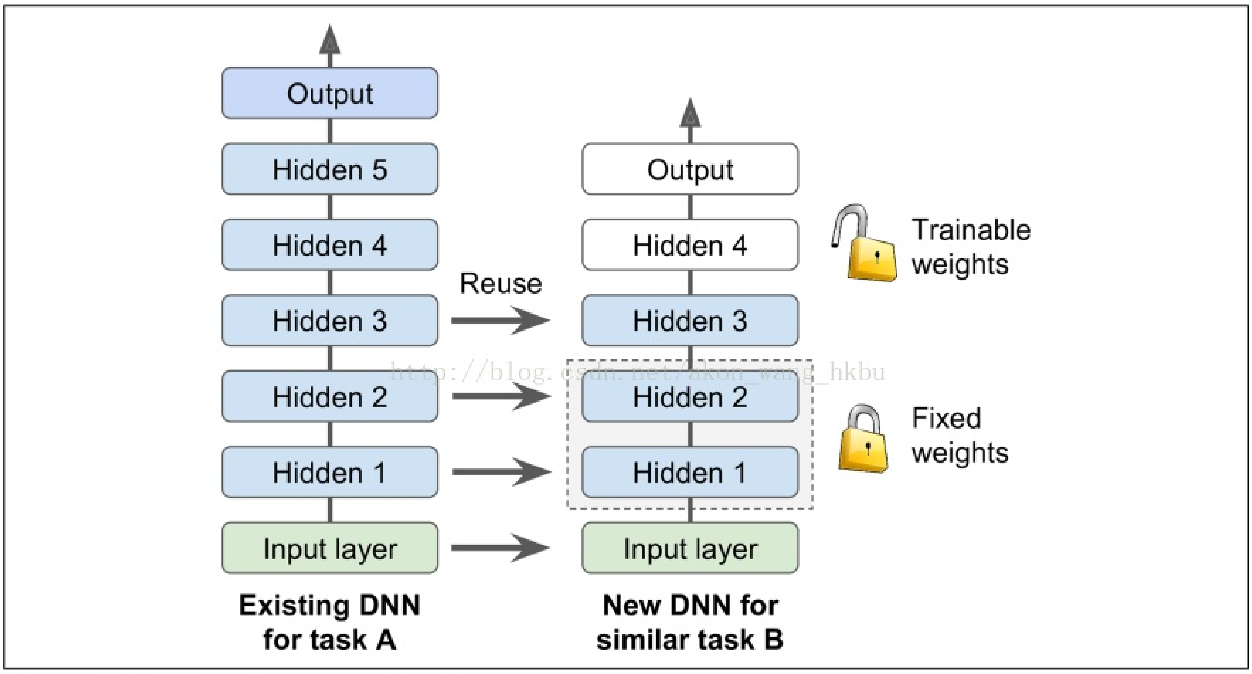


Figure 11-4. Reusing pretrained layers

如果新任务的输入图像与原始任务中使用的输入图像的大小不一致，则必须添加预处理步骤以将其大小调整为原始模型的预期大小。 更一般地说，如果输入具有类似的低级层次的特征，则迁移学习将很好地工作。

复用 TensorFlow 模型

如果原始模型使用 TensorFlow 进行训练，则可以简单地将其恢复并在新任务上进行训练：

```
[...] # construct the original model
```

```
with tf.Session() as sess:
    saver.restore(sess, "./my_model_final.ckpt")
    # continue training the model...
```

完整代码:

```
n_inputs = 28 * 28  # MNIST
n_hidden1 = 300
n_hidden2 = 50
n_hidden3 = 50
n_hidden4 = 50
n_outputs = 10

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")

with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu, name="hidden1")
    hidden2 = tf.layers.dense(hidden1, n_hidden2, activation=tf.nn.relu, name="hidden2"
)
    hidden3 = tf.layers.dense(hidden2, n_hidden3, activation=tf.nn.relu, name="hidden3"
)
    hidden4 = tf.layers.dense(hidden3, n_hidden4, activation=tf.nn.relu, name="hidden4"
)
    hidden5 = tf.layers.dense(hidden4, n_hidden5, activation=tf.nn.relu, name="hidden5"
)
    logits = tf.layers.dense(hidden5, n_outputs, name="outputs")

with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")

with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32), name="accuracy")

learning_rate = 0.01
threshold = 1.0

optimizer = tf.train.GradientDescentOptimizer(learning_rate)
grads_and_vars = optimizer.compute_gradients(loss)
capped_gvs = [(tf.clip_by_value(grad, -threshold, threshold), var)
              for grad, var in grads_and_vars]
training_op = optimizer.apply_gradients(capped_gvs)

init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

```
with tf.Session() as sess:
    saver.restore(sess, "./my_model_final.ckpt")

    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        accuracy_val = accuracy.eval(feed_dict={X: mnist.test.images,
                                                y: mnist.test.labels})
        print(epoch, "Test accuracy:", accuracy_val)

    save_path = saver.save(sess, "./my_new_model_final.ckpt")
```

但是，一般情况下，您只需要重新使用原始模型的一部分（就像我们将要讨论的那样）。一个简单的解决方案是将 `Saver` 配置为仅恢复原始模型中的一部分变量。例如，下面的代码只恢复隐藏的层1,2和3：

```
n_inputs = 28 * 28  # MNIST
n_hidden1 = 300 # reused
n_hidden2 = 50  # reused
n_hidden3 = 50  # reused
n_hidden4 = 20  # new!
n_outputs = 10  # new!

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")

with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu, name="hidden1")
   # reused
    hidden2 = tf.layers.dense(hidden1, n_hidden2, activation=tf.nn.relu, name="hidden2"
) # reused
    hidden3 = tf.layers.dense(hidden2, n_hidden3, activation=tf.nn.relu, name="hidden3"
) # reused
    hidden4 = tf.layers.dense(hidden3, n_hidden4, activation=tf.nn.relu, name="hidden4"
) # new!
    logits = tf.layers.dense(hidden4, n_outputs, name="outputs")
   # new!

with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")

with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32), name="accuracy")

with tf.name_scope("train"):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    training_op = optimizer.minimize(loss)
```

```
[...] # build new model with the same definition as before for hidden layers 1-3
```

```
reuse_vars = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES,
                               scope="hidden[123]") # regular expression
reuse_vars_dict = dict([(var.op.name, var) for var in reuse_vars])
restore_saver = tf.train.Saver(reuse_vars_dict) # to restore layers 1-3

init = tf.global_variables_initializer()
saver = tf.train.Saver()

with tf.Session() as sess:
    init.run()
    restore_saver.restore(sess, "./my_model_final.ckpt")

    for epoch in range(n_epochs):                                    # not shown in
the book
        for iteration in range(mnist.train.num_examples // batch_size): # not shown
            X_batch, y_batch = mnist.train.next_batch(batch_size)      # not shown
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})  # not shown
        accuracy_val = accuracy.eval(feed_dict={X: mnist.test.images,  # not shown
                                                y: mnist.test.labels}) # not shown
        print(epoch, "Test accuracy:", accuracy_val)                   # not shown

    save_path = saver.save(sess, "./my_new_model_final.ckpt")
```

首先我们建立新的模型，确保复制原始模型的隐藏层 1 到 3。我们还创建一个节点来初始化所有变量。然后我们得到刚刚用 `trainable = True`（这是默认值）创建的所有变量的列表，我们只保留那些范围与正则表达式 `hidden [123]` 相匹配的变量（即，我们得到所有可训练的

隐藏层 1 到 3 中的变量）。 接下来，我们创建一个字典，将原始模型中每个变量的名称映射到新模型中的名称（通常需要保持完全相同的名称）。 然后，我们创建一个 `Saver` ，它将只恢复这些变量，并且创建另一个 `Saver` 来保存整个新模型，而不仅仅是第 1 层到第 3 层。然后，我们开始一个会话并初始化模型中的所有变量，然后从原始模型的层 1 到 3中恢复变量值。最后，我们在新任务上训练模型并保存。

任务越相似，您可以重复使用的层越多（从较低层开始）。 对于非常相似的任务，您可以尝试保留所有隐藏的层，只替换输出层。

复用来自其它框架的模型

如果模型是使用其他框架进行训练的，则需要手动加载权重（例如，如果使用 Theano 训练，则使用 Theano 代码），然后将它们分配给相应的变量。 这可能是相当乏味的。 例如，下面的代码显示了如何复制使用另一个框架训练的模型的第一个隐藏层的权重和偏置：

```
original_w = [...] # Load the weights from the other framework
original_b = [...] # Load the biases from the other framework

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
[...] # # Build the rest of the model

# Get a handle on the variables created by fully_connected()
with tf.variable_scope("", default_name="", reuse=True):  # root scope
    hidden1_weights = tf.get_variable("hidden1/weights")
    hidden1_biases = tf.get_variable("hidden1/biases")

# Create nodes to assign arbitrary values to the weights and biases
original_weights = tf.placeholder(tf.float32, shape=(n_inputs, n_hidden1))
original_biases = tf.placeholder(tf.float32, shape=(n_hidden1))
assign_hidden1_weights = tf.assign(hidden1_weights, original_weights)
assign_hidden1_biases = tf.assign(hidden1_biases, original_biases)

init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    sess.run(assign_hidden1_weights, feed_dict={original_weights: original_w})
    sess.run(assign_hidden1_biases, feed_dict={original_biases: original_b})
    [...] # Train the model on your new task
```


冻结较低层

第一个 DNN 的较低层可能已经学会了检测图片中的低级特征，这将在两个图像分类任务中有用，因此您可以按照原样重新使用这些层。 在训练新的 DNN 时，“冻结”权重通常是一个好主意：如果较低层权重是固定的，那么较高层权重将更容易训练（因为他们不需要学习一个移

动的目标）。要在训练期间冻结较低层，最简单的解决方案是给优化器列出要训练的变量，不包括来自较低层的变量：

```
train_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                               scope="hidden[34]|outputs")
training_op = optimizer.minimize(loss, var_list=train_vars)
```

第一行获得隐藏层 3 和 4 以及输出层中所有可训练变量的列表。这留下了隐藏层 1 和 2 中的变量。接下来，我们将这个受限制的可列表变量列表提供给 `optimizer` 的 `minimize()` 函数。当当！现在，层 1 和层 2 被冻结：在训练过程中不会发生变化（通常称为冻结层）。

缓存冻结层

由于冻结层不会改变，因此可以为每个训练实例缓存最上面的冻结层的输出。由于训练贯穿整个数据集很多次，这将给你一个巨大的速度提升，因为每个训练实例只需要经过一次冻结层（而不是每个迭代一次）。例如，你可以先运行整个训练集（假设你有足够的内存）：

```
hidden2_outputs = sess.run(hidden2, feed_dict={X: X_train})
```

然后在训练过程中，不再对训练实例建立批次，而是从隐藏层2的输出建立批次，并将它们提供给训练操作：

```
import numpy as np

n_epochs = 100
n_batches = 500

for epoch in range(n_epochs):
    shuffled_idx = rnd.permutation(len(hidden2_outputs))
    hidden2_batches = np.array_split(hidden2_outputs[shuffled_idx], n_batches)
    y_batches = np.array_split(y_train[shuffled_idx], n_batches)
    for hidden2_batch, y_batch in zip(hidden2_batches, y_batches):
        sess.run(training_op, feed_dict={hidden2: hidden2_batch, y: y_batch})
```

最后一行运行先前定义的训练操作（冻结层 1 和 2），并从第二个隐藏层（以及该批次的目标）为其输出一批输出。因为我们给 TensorFlow 隐藏层 2 的输出，所以它不会去评估它（或者它所依赖的任何节点）。

调整，删除或替换较高层

原始模型的输出层通常应该被替换，因为对于新的任务来说，最有可能没有用处，甚至可能没有适合新任务的输出数量。

类似地，原始模型的较高隐藏层不太可能像较低层一样有用，因为对于新任务来说最有用的高层特征可能与对原始任务最有用的高层特征明显不同。你需要找到正确的层数来复用。

尝试先冻结所有复制的层，然后训练模型并查看它是如何执行的。 然后尝试解冻一个或两个较高隐藏层，让反向传播调整它们，看看性能是否提高。 您拥有的训练数据越多，您可以解冻的层数就越多。

如果仍然无法获得良好的性能，并且您的训练数据很少，请尝试删除顶部的隐藏层，并再次冻结所有剩余的隐藏层。 您可以迭代，直到找到正确的层数重复使用。 如果您有足够的训练数据，您可以尝试替换顶部的隐藏层，而不是丢掉它们，甚至可以添加更多的隐藏层。

Model Zoos

你在哪里可以找到一个类似于你想要解决的任务训练的神经网络？ 首先看看显然是在你自己的模型目录。 这是保存所有模型并组织它们的一个很好的理由，以便您以后可以轻松地检索它们。 另一个选择是在模型动物园中搜索。 许多人为了各种不同的任务而训练机器学习模型，并且善意地向公众发布预训练模型。

TensorFlow 在 https://github.com/tensorflow/models 中有自己的模型动物园。 特别是，它包含了大多数最先进的图像分类网络，如 VGG，Inception 和 ResNet（参见第 13 章，检查 `model/slim` 目录），包括代码，预训练模型和 工具来下载流行的图像数据集。

另一个流行的模型动物园是 Caffe 模型动物园。 它还包含许多在各种数据集（例如，ImageNet，Places 数据库，CIFAR10 等）上训练的计算机视觉模型（例如，LeNet，AlexNet，ZFNet，GoogLeNet，VGGNet，开始）。 Saumitro Dasgupta 写了一个转换器，可以在 https://github.com/ethereon/caffetensorflow。

无监督的预训练

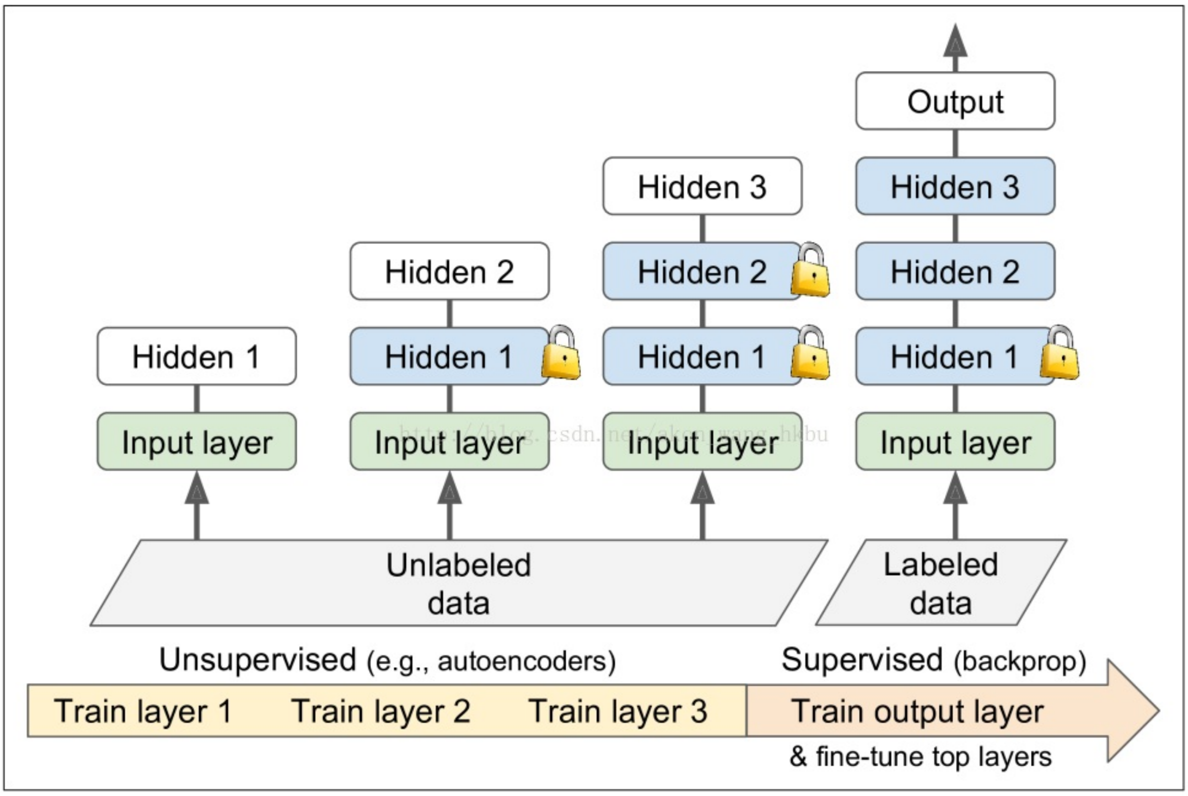


Figure 11-5. Unsupervised pretraining

假设你想要解决一个复杂的任务，你没有太多的标记的训练数据，但不幸的是，你不能找到一个类似的任务训练模型。 不要失去所有希望！ 首先，你当然应该尝试收集更多的有标签的训练数据，但是如果这太难或太昂贵，你仍然可以进行无监督的训练（见图 11-5）。 也就是说，如果你有很多未标记的训练数据，你可以尝试逐层训练层，从最低层开始，然后上升，使用无监督的特征检测算法，如限制玻尔兹曼机（RBM；见附录 E）或自动编码器（见第 15 章）。 每个层都被训练成先前训练过的层的输出（除了被训练的层之外的所有层都被冻结）。 一旦所有层都以这种方式进行了训练，就可以使用监督式学习（即反向传播）对网络进行微调。

这是一个相当漫长而乏味的过程，但通常运作良好。 实际上，这是 Geoffrey Hinton 和他的团队在 2006 年使用的技术，导致了神经网络的复兴和深度学习的成功。 直到 2010 年，无监督预训练（通常使用 RBM）是深度网络的标准，只有在梯度消失问题得到缓解之后，纯训练 DNN 才更为普遍。 然而，当您有一个复杂的任务需要解决时，无监督训练（现在通常使用自动编码器而不是 RBM）仍然是一个很好的选择，没有类似的模型可以重复使用，而且标记的训练数据很少，但是大量的未标记的训练数据。（另一个选择是提出一个监督的任务，您可以轻松地收集大量标记的训练数据，然后使用迁移学习，如前所述。 例如，如果要训练一个模型来识别图片中的朋友，你可以在互联网上下载数百万张脸并训练一个分类器来检测两张脸是否相同，然后使用此分类器将新图片与你朋友的每张照片做比较。）

在辅助任务上预训练

最后一种选择是在辅助任务上训练第一个神经网络，您可以轻松获取或生成标记的训练数据，然后重新使用该网络的较低层来完成您的实际任务。 第一个神经网络的较低层将学习可能被第二个神经网络重复使用的特征检测器。

例如，如果你想建立一个识别面孔的系统，你可能只有几个人的照片 - 显然不足以训练一个好的分类器。 收集每个人的数百张照片将是不实际的。 但是，您可以在互联网上收集大量随机人员的照片，并训练第一个神经网络来检测两张不同的照片是否属于同一个人。 这样的网络将学习面部优秀的特征检测器，所以重复使用它的较低层将允许你使用很少的训练数据来训练一个好的面部分类器。

收集没有标签的训练样本通常是相当便宜的，但标注它们却相当昂贵。 在这种情况下，一种常见的技术是将所有训练样例标记为“好”，然后通过破坏好的训练样例产生许多新的训练样例，并将这些样例标记为“坏”。然后，您可以训练第一个神经网络 将实例分类为好或不好。例如，您可以下载数百万个句子，将其标记为“好”，然后在每个句子中随机更改一个单词，并将结果语句标记为“不好”。如果神经网络可以告诉“The dog sleeps”是好的句子，但“The dog they”是坏的，它可能知道相当多的语言。 重用其较低层可能有助于许多语言处理任务。

另一种方法是训练第一个网络为每个训练实例输出一个分数，并使用一个损失函数确保一个好的实例的分数大于一个坏实例的分数至少一定的边际。 这被称为最大边际学习.

更快的优化器

训练一个非常大的深度神经网络可能会非常缓慢。 到目前为止，我们已经看到了四种加速训练的方法（并且达到更好的解决方案）：对连接权重应用良好的初始化策略，使用良好的激活函数，使用批量规范化以及重用预训练网络的部分。 另一个巨大的速度提升来自使用比普通渐变下降优化器更快的优化器。 在本节中，我们将介绍最流行的：动量优化，Nesterov 加速梯度，AdaGrad，RMSProp，最后是 Adam 优化。

剧透：本节的结论是，您几乎总是应该使用 `Adam_optimization` ，所以如果您不关心它是如何工作的，只需使用 `AdamOptimizer` 替换您的 `GradientDescentOptimizer` ，然后跳到下一节！ 只需要这么小的改动，训练通常会快几倍。 但是，Adam 优化确实有三个可以调整的超参数（加上学习率）。 默认值通常工作的不错，但如果您需要调整它们，知道他们怎么实现的可能会有帮助。 Adam 优化结合了来自其他优化算法的几个想法，所以先看看这些算法是有用的。

动量优化

想象一下，一个保龄球在一个光滑的表面上平缓的斜坡上滚动：它会缓慢地开始，但是它会很快地达到最终的速度（如果有一些摩擦或空气阻力的话）。 这是 Boris Polyak 在 1964 年提出的动量优化背后的一个非常简单的想法。相比之下，普通的梯度下降只需要沿着斜坡进行小的有规律的下降步骤，所以需要更多的时间才能到达底部。

回想一下，梯度下降只是通过直接减去损失函数 `J(θ)` 相对于权重 `θ` 的梯度，乘以学习率 `η` 来更新权重 `θ` 。 方程是：$\theta := \theta - \eta \nabla_\theta J(\theta)$。它不关心早期的梯度是什么。 如果局部梯度很小，则会非常缓慢。

Equation 11-4. Momentum algorithm

$$1.\quad \mathbf{m} \leftarrow \beta \mathbf{m} + \eta \nabla_\theta J(\theta)$$

$$2.\quad \theta \leftarrow \theta - \mathbf{m}$$

动量优化很关心以前的梯度：在每次迭代时，它将动量矢量 `m` （乘以学习率 `η` ）与局部梯度相加，并且通过简单地减去该动量矢量来更新权重（参见公式 11-4）。 换句话说，梯度用作加速度，不用作速度。 为了模拟某种摩擦机制，避免动量过大，该算法引入了一个新的超参数 `β` ，简称为动量，它必须设置在 0（高摩擦）和 1（无摩擦）之间。 典型的动量值是 0.9。

您可以很容易地验证，如果梯度保持不变，则最终速度（即，权重更新的最大大小）等于该梯度乘以学习率 `η` 乘以 `1/(1-β)` 。 例如，如果 `β = 0.9` ，则最终速度等于学习率的梯度乘以 10 倍，因此动量优化比梯度下降快 10 倍！ 这使动量优化比梯度下降快得多。 特别是，我们在第四章中看到，当输入量具有非常不同的尺度时，损失函数看起来像一个细长的碗（见图 4-7）。 梯度下降速度很快，但要花很长的时间才能到达底部。 相反，动量优化会越来越快地滚下山谷底部，直到达到底部（最佳）。

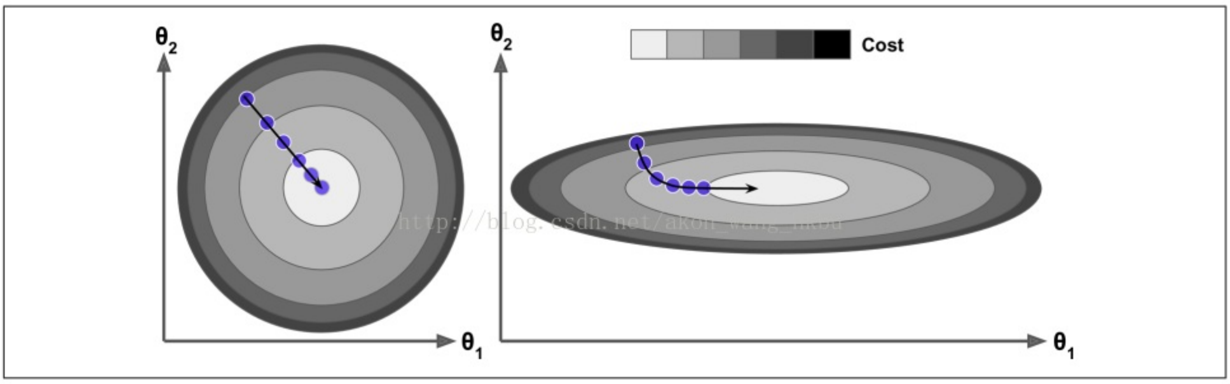


Figure 4-7. Gradient Descent with and without feature scaling

在不使用批标准化的深层神经网络中，较高层往往会得到具有不同的尺度的输入，所以使用动量优化会有很大的帮助。 它也可以帮助滚过局部最优值。

由于动量的原因，优化器可能会超调一些，然后再回来，再次超调，并在稳定在最小值之前多次振荡。 这就是为什么在系统中有一点摩擦的原因之一：它消除了这些振荡，从而加速了收敛。

在 TensorFlow 中实现动量优化是一件简单的事情：只需用 `MomentumOptimizer` 替换 `GradientDescentOptimizer` ，然后躺下来赚钱！

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                       momentum=0.9)
```


动量优化的一个缺点是它增加了另一个超参数来调整。 然而，0.9 的动量值通常在实践中运行良好，几乎总是比梯度下降快。

Nesterov 加速梯度

Yurii Nesterov 在 1983 年提出的动量优化的一个小变体几乎总是比普通的动量优化更快。Nesterov 动量优化或 Nesterov 加速梯度（Nesterov Accelerated Gradient，NAG）的思想是测量损失函数的梯度不是在局部位置，而是在动量方向稍微靠前（见公式 11-5）。 与普通的动量优化的唯一区别在于梯度是在 `θ+βm` 而不是在 `θ` 处测量的。

Equation 11-5. Nesterov Accelerated Gradient algorithm

1. $\mathbf{m} \leftarrow \beta\mathbf{m} + \eta\nabla_\theta J(\theta + \beta\mathbf{m})$
2. $\theta \leftarrow \theta - \mathbf{m}$

这个小小的调整是可行的，因为一般来说，动量矢量将指向正确的方向（即朝向最优方向），所以使用在该方向上测得的梯度稍微更精确，而不是使用 原始位置的梯度，如图11-6所示（其中 `∇1` 代表在起点 `θ` 处测量的损失函数的梯度， `∇2` 代表位于 `θ+βm` 的点处的梯度）。

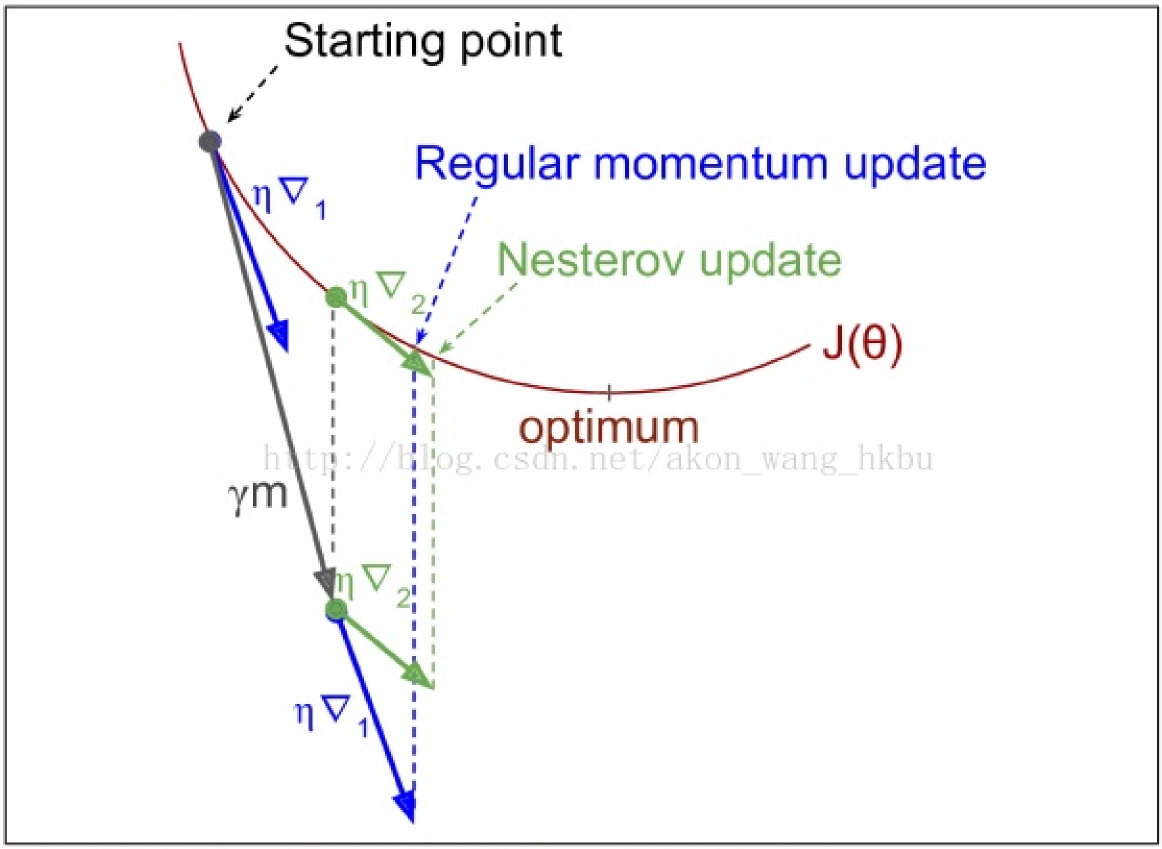


Figure 11-6. Regular versus Nesterov Momentum optimization

正如你所看到的，Nesterov 更新稍微靠近最佳值。 过了一段时间，这些小的改进加起来，NAG 最终比常规的动量优化快得多。 此外，请注意，当动量推动权重横跨山谷时，▽1继续推进越过山谷，而▽2推回山谷的底部。 这有助于减少振荡，从而更快地收敛。

与常规的动量优化相比，NAG 几乎总能加速训练。 要使用它，只需在创建 `MomentumOptimizer` 时设置 `use_nesterov = True` ：

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                       momentum=0.9, use_nesterov=True)
```

AdaGrad

再次考虑细长碗的问题：梯度下降从最陡峭的斜坡快速下降，然后缓慢地下到谷底。 如果算法能够早期检测到这个问题并且纠正它的方向来指向全局最优点，那将是非常好的。

AdaGrad 算法通过沿着最陡的维度缩小梯度向量来实现这一点（见公式 11-6）：

Equation 11-6. AdaGrad algorithm

$$1.\quad \mathbf{s} \leftarrow \mathbf{s} + \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$$

$$2.\quad \theta \leftarrow \theta - \eta\, \nabla_\theta J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}$$

第一步将梯度的平方累加到矢量 `s` 中（`⊗` 符号表示单元乘法）。 这个向量化形式相当于向量 `s` 的每个元素 `si` 计算 $s_i := s_i + (\partial J(\theta)/\partial\theta_i)^2$。换一种说法，每个 s_i 累加损失函数对参数 θ_i 的偏导数的平方。 如果损失函数沿着第 `i` 维陡峭，则在每次迭代时，s_i 将变得越来越大。

第二步几乎与梯度下降相同，但有一个很大的不同：梯度矢量按比例缩小 $\sqrt{s + \epsilon}$ （`⊘` 符号表示元素分割，`ε` 是避免被零除的平滑项，通常设置为 10^{-10}。 这个矢量化的形式相当于计算 $\theta_i \leftarrow \theta_i - \eta\, \partial/\partial\theta_i\, J(\theta)/\sqrt{s_i + \epsilon}$ 对于所有参数 θ_i（同时）。

简而言之，这种算法会降低学习速度，但对于陡峭的尺寸，其速度要快于具有温和的斜率的尺寸。 这被称为自适应学习率。 它有助于将更新的结果更直接地指向全局最优（见图 11-7）。 另一个好处是它不需要那么多的去调整学习率超参数 `η` 。

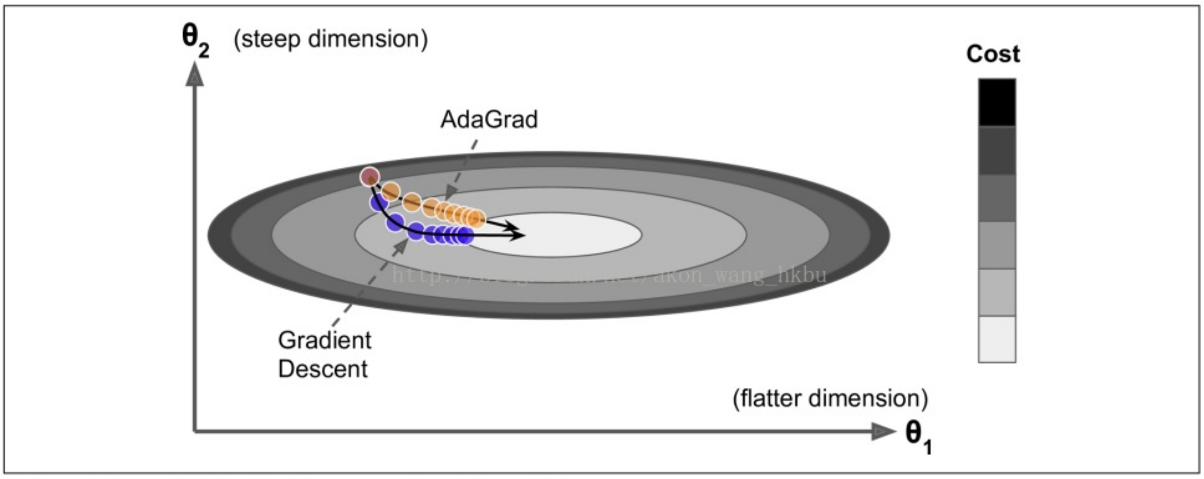


Figure 11-7. AdaGrad versus Gradient Descent

对于简单的二次问题，AdaGrad 经常表现良好，但不幸的是，在训练神经网络时，它经常停止得太早。 学习率被缩减得太多，以至于在达到全局最优之前，算法完全停止。 所以，即使 TensorFlow 有一个 `AdagradOptimizer` ，你也不应该用它来训练深度神经网络（虽然对线性回归这样简单的任务可能是有效的）。

RMSProp

尽管 AdaGrad 的速度变慢了一点，并且从未收敛到全局最优，但是 RMSProp 算法通过仅累积最近迭代（而不是从训练开始以来的所有梯度）的梯度来修正这个问题。 它通过在第一步中使用指数衰减来实现（见公式 11-7）。

Equation 11-7. RMSProp algorithm

$$1.\quad \mathbf{s} \leftarrow \beta \mathbf{s} + (1-\beta)\nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$$

$$2.\quad \theta \leftarrow \theta - \eta\, \nabla_\theta J(\theta) \oslash \sqrt{\mathbf{s}+\epsilon}$$

他的衰变率 `β` 通常设定为 0.9。 是的，它又是一个新的超参数，但是这个默认值通常运行良好，所以你可能根本不需要调整它。

正如您所料，TensorFlow 拥有一个 `RMSPropOptimizer` 类：

```
optimizer = tf.train.RMSPropOptimizer(learning_rate=learning_rate,
                                      momentum=0.9, decay=0.9, epsilon=1e-10)
```

除了非常简单的问题，这个优化器几乎总是比 AdaGrad 执行得更好。 它通常也比动量优化和 Nesterov 加速梯度表现更好。 事实上，这是许多研究人员首选的优化算法，直到 Adam 优化出现。

Adam 优化

Adam，代表自适应矩估计，结合了动量优化和 RMSProp 的思想：就像动量优化一样，它追踪过去梯度的指数衰减平均值，就像 RMSProp 一样，它跟踪过去平方梯度的指数衰减平均值（见方程式 11-8）。

Equation 11-8. Adam algorithm

$$1.\quad \mathbf{m} \leftarrow \beta_1 \mathbf{m} + (1-\beta_1)\nabla_\theta J(\theta)$$

$$2.\quad \mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1-\beta_2)\nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$$

$$3.\quad \mathbf{m} \leftarrow \frac{\mathbf{m}}{1-{\beta_1}^T}$$

$$4.\quad \mathbf{s} \leftarrow \frac{\mathbf{s}}{1-{\beta_2}^T}$$

$$5.\quad \theta \leftarrow \theta - \eta\, \mathbf{m} \oslash \sqrt{\mathbf{s}+\epsilon}$$

T 代表迭代次数（从 1 开始）。

如果你只看步骤 1, 2 和 5，你会注意到 Adam 与动量优化和 RMSProp 的相似性。 唯一的区别是第 1 步计算指数衰减的平均值，而不是指数衰减的和，但除了一个常数因子（衰减平均值只是衰减和的 `1 - β1` 倍）之外，它们实际上是等效的。 步骤 3 和步骤 4 是一个技术细节：由于 `m` 和 `s` 初始化为 0，所以在训练开始时它们会偏向0，所以这两步将在训练开始时帮助提高 `m` 和 `s` 。

动量衰减超参数 `β1` 通常初始化为 0.9，而缩放衰减超参数 `β2` 通常初始化为 0.999。 如前所述，平滑项 `ε` 通常被初始化为一个很小的数，例如 10^{-8} 。这些是 TensorFlow 的 `AdamOptimizer` 类的默认值，所以你可以简单地使用：

```
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
```

实际上，由于 Adam 是一种自适应学习率算法（如 AdaGrad 和 RMSProp），所以对学习率超参数 `η` 的调整较少。 您经常可以使用默认值 `η= 0.001` ，使 Adam 更容易使用相对于梯度下降。

迄今为止所讨论的所有优化技术都只依赖于一阶偏导数（雅可比矩阵）。 优化文献包含基于二阶偏导数（海森矩阵）的惊人算法。 不幸的是，这些算法很难应用于深度神经网络，因为每个输出有 `n ^ 2` 个海森值（其中 `n` 是参数的数量），而不是每个输出只有 `n` 个雅克比值。由于 DNN 通常具有数以万计的参数，二阶优化算法通常甚至不适合内存，甚至在他们这样做时，计算海森矩阵也是太慢了。

训练稀疏模型

所有刚刚提出的优化算法都会产生密集的模型，这意味着大多数参数都是非零的。 如果你在运行时需要一个非常快速的模型，或者如果你需要它占用较少的内存，你可能更喜欢用一个稀疏模型来代替。

实现这一点的一个微不足道的方法是像平常一样训练模型，然后摆脱微小的权重（将它们设置为 0）。

另一个选择是在训练过程中应用强 l1 正则化，因为它会推动优化器尽可能多地消除权重（如第 4 章关于 Lasso 回归的讨论）。

但是，在某些情况下，这些技术可能仍然不足。 最后一个选择是应用双重平均，通常称为遵循正则化领导者（FTRL），一种由尤里·涅斯捷罗夫（Yurii Nesterov）提出的技术。 当与 l1 正则化一起使用时，这种技术通常导致非常稀疏的模型。 TensorFlow 在 `FTRLOptimizer` 类中实现称为 FTRL-Proximal 的 FTRL 变体。

学习率调整

找到一个好的学习速度可能会非常棘手。 如果设置太高，训练实际上可能偏离（如我们在第 4 章）。 如果设置得太低，训练最终会收敛到最佳状态，但这需要很长时间。 如果将其设置得太高，开始的进度会非常快，但最终会在最优解周围跳动，永远不会安顿下来（除非您使用自适应学习率优化算法，如 AdaGrad，RMSProp 或 Adam，但是 即使这样可能需要时间来解决）。 如果您的计算预算有限，那么您可能必须在正确收敛之前中断训练，产生次优解决方案（参见图 11-8）。

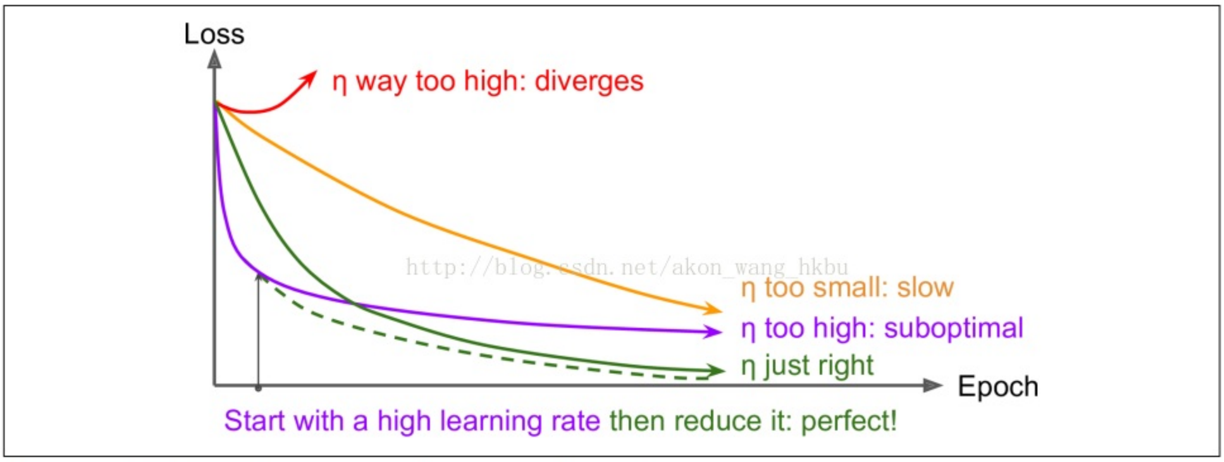


Figure 11-8. Learning curves for various learning rates η

通过使用各种学习率和比较学习曲线，在几个迭代内对您的网络进行多次训练，您也许能够找到相当好的学习率。 理想的学习率将会快速学习并收敛到良好的解决方案。

然而，你可以做得比不断的学习率更好：如果你从一个高的学习率开始，然后一旦它停止快速的进步就减少它，你可以比最佳的恒定学习率更快地达到一个好的解决方案。 有许多不同的策略，以减少训练期间的学习率。 这些策略被称为学习率调整（我们在第 4 章中简要介绍了这个概念），其中最常见的是：

预定的分段恒定学习率:

例如，首先将学习率设置为 $_0 = 0.1$，然后在 50 个迭代之后将学习率设置为 $_1 = 0.001$。虽然这个解决方案可以很好地工作，但是通常需要弄清楚正确的学习速度以及何时使用它们。

性能调度:

每 N 步测量验证误差（就像提前停止一样），当误差下降时，将学习率降低一个因子 `λ` 。

指数调度:

将学习率设置为迭代次数 `t` 的函数：$\eta(t) = \eta_0\ 10^{-t/r}$。这很好，但它需要调整 `η0` 和 `r` 。 学习率将由每 `r` 步下降 10 倍。

幂调度:

设学习率为 $\eta(t) = \eta_0\ (1 + t/r)^{-c}$。 超参数 `c` 通常被设置为 1。这与指数调度类似，但是学习率下降要慢得多。

Andrew Senior 等2013年的论文。 比较了使用动量优化训练深度神经网络进行语音识别时一些最流行的学习率调整的性能。 作者得出结论：在这种情况下，性能调度和指数调度都表现良好，但他们更喜欢指数调度，因为它实现起来比较简单，容易调整，收敛速度略快于最佳解决方案。

使用 TensorFlow 实现学习率调整非常简单：

```
    initial_learning_rate = 0.1
    decay_steps = 10000
    decay_rate = 1/10
    global_step = tf.Variable(0, trainable=False, name="global_step")
    learning_rate = tf.train.exponential_decay(initial_learning_rate, global_step,
                                               decay_steps, decay_rate)
    optimizer = tf.train.MomentumOptimizer(learning_rate, momentum=0.9)
    training_op = optimizer.minimize(loss, global_step=global_step)
```

设置超参数值后，我们创建一个不可训练的变量 `global_step` （初始化为 0）以跟踪当前的训练迭代次数。 然后我们使用 TensorFlow 的 `exponential_decay()` 函数来定义指数衰减的学习率（ `η0= 0.1` 和 `r = 10,000` ）。 接下来，我们使用这个衰减的学习率创建一个优化器（在这个例子中是一个 `MomentumOptimizer` ）。 最后，我们通过调用优化器的 `minimize()` 方法来创建训练操作；因为我们将 `global_step` 变量传递给它，所以请注意增加它。 就是这样！

由于 AdaGrad，RMSProp 和 Adam 优化自动降低了训练期间的学习率，因此不需要添加额外的学习率调整。 对于其他优化算法，使用指数衰减或性能调度可显著加速收敛。

完整代码:

```
n_inputs = 28 * 28  # MNIST
n_hidden1 = 300
n_hidden2 = 50
n_outputs = 10

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")

with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu, name="hidden1")
    hidden2 = tf.layers.dense(hidden1, n_hidden2, activation=tf.nn.relu, name="hidden2"
)
    logits = tf.layers.dense(hidden2, n_outputs, name="outputs")

with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")

with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32), name="accuracy")
```

```
with tf.name_scope("train"):       # not shown in the book
    initial_learning_rate = 0.1
    decay_steps = 10000
    decay_rate = 1/10
    global_step = tf.Variable(0, trainable=False, name="global_step")
    learning_rate = tf.train.exponential_decay(initial_learning_rate, global_step,
                                               decay_steps, decay_rate)
    optimizer = tf.train.MomentumOptimizer(learning_rate, momentum=0.9)
    training_op = optimizer.minimize(loss, global_step=global_step)
```

```
init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

```
n_epochs = 5
batch_size = 50

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        accuracy_val = accuracy.eval(feed_dict={X: mnist.test.images,
                                                y: mnist.test.labels})
        print(epoch, "Test accuracy:", accuracy_val)

    save_path = saver.save(sess, "./my_model_final.ckpt")
```

通过正则化避免过拟合

有四个参数，我可以拟合一个大象，五个我可以让他摆动他的象鼻。

—— John von Neumann,cited by Enrico Fermi in Nature 427

深度神经网络通常具有数以万计的参数，有时甚至是数百万。 有了这么多的参数，网络拥有难以置信的自由度，可以适应各种复杂的数据集。 但是这个很大的灵活性也意味着它很容易过拟合训练集。

有了数以百万计的参数，你可以适应整个动物园。 在本节中，我们将介绍一些最流行的神经网络正则化技术，以及如何用 TensorFlow 实现它们：早期停止，l1 和 l2 正则化，drop out，最大范数正则化和数据增强。

早期停止

为避免过度拟合训练集，一个很好的解决方案就是尽早停止训练（在第 4 章中介绍）：只要在训练集的性能开始下降时中断训练。

与 TensorFlow 实现方法之一是评估其对设置定期（例如，每 50 步）验证模型，并保存一个“winner”的快照，如果它优于以前“winner”的快照。 计算自上次“winner”快照保存以来的步数，并在达到某个限制时（例如 2000 步）中断训练。 然后恢复最后的“winner”快照。

虽然早期停止在实践中运行良好，但是通过将其与其他正则化技术相结合，您通常可以在网络中获得更高的性能。

L1 和 L2 正则化

就像你在第 4 章中对简单线性模型所做的那样，你可以使用 l1 和 l2 正则化约束一个神经网络的连接权重（但通常不是它的偏置）。

使用 TensorFlow 做到这一点的一种方法是简单地将适当的正则化项添加到您的损失函数中。例如，假设您只有一个权重为 `weights1` 的隐藏层和一个权重为 `weight2` 的输出层，那么您可以像这样应用 l1 正则化：

我们可以将正则化函数传递给 `tf.layers.dense()` 函数，该函数将使用它来创建计算正则化损失的操作，并将这些操作添加到正则化损失集合中。 开始和上面一样：

```
n_inputs = 28 * 28  # MNIST
n_hidden1 = 300
n_hidden2 = 50
n_outputs = 10

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")
```

接下来，我们将使用 Python `partial()` 函数来避免一遍又一遍地重复相同的参数。 请注意，我们设置了内核正则化参数（正则化函数有 `l1_regularizer()`，`l2_regularizer()`，`l1_l2_regularizer()`）：

```
scale = 0.001
```

```
my_dense_layer = partial(
    tf.layers.dense, activation=tf.nn.relu,
    kernel_regularizer=tf.contrib.layers.l1_regularizer(scale))

with tf.name_scope("dnn"):
    hidden1 = my_dense_layer(X, n_hidden1, name="hidden1")
    hidden2 = my_dense_layer(hidden1, n_hidden2, name="hidden2")
    logits = my_dense_layer(hidden2, n_outputs, activation=None,
                            name="outputs")
```

该代码创建了一个具有两个隐藏层和一个输出层的神经网络，并且还在图中创建节点以计算与每个层的权重相对应的 l1 正则化损失。TensorFlow 会自动将这些节点添加到包含所有正则化损失的特殊集合中。 您只需要将这些正则化损失添加到您的整体损失中，如下所示：

接下来，我们必须将正则化损失加到基本损失上：

```
with tf.name_scope("loss"):                                     # not shown in the book

    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(  # not shown
        labels=y, logits=logits)                                # not shown
    base_loss = tf.reduce_mean(xentropy, name="avg_xentropy")   # not shown
    reg_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
    loss = tf.add_n([base_loss] + reg_losses, name="loss")
```

其余的和往常一样：

```
with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32), name="accuracy")

learning_rate = 0.01

with tf.name_scope("train"):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    training_op = optimizer.minimize(loss)

init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

```
n_epochs = 20
batch_size = 200

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        accuracy_val = accuracy.eval(feed_dict={X: mnist.test.images,
                                                y: mnist.test.labels})
        print(epoch, "Test accuracy:", accuracy_val)

    save_path = saver.save(sess, "./my_model_final.ckpt")
```

不要忘记把正则化的损失加在你的整体损失上，否则就会被忽略。

Dropout

深度神经网络最流行的正则化技术可以说是 dropout。 它由 GE Hinton 于 2012 年提出，并在 Nitish Srivastava 等人的论文中进一步详细描述，并且已被证明是非常成功的：即使是最先进的神经网络，仅仅通过增加丢失就可以提高1-2%的准确度。 这听起来可能不是很多，但是当一个模型已经具有 95% 的准确率时，获得 2% 的准确度提升意味着将误差率降低近 40%（从 5% 误差降至大约 3%）。

这是一个相当简单的算法：在每个训练步骤中，每个神经元（包括输入神经元，但不包括输出神经元）都有一个暂时“丢弃”的概率 `p` ，这意味着在这个训练步骤中它将被完全忽略， 在下一步可能会激活（见图 11-9）。 超参数 `p` 称为丢失率，通常设为 50%。 训练后，神经元不会再下降。 这就是全部（除了我们将要讨论的技术细节）。

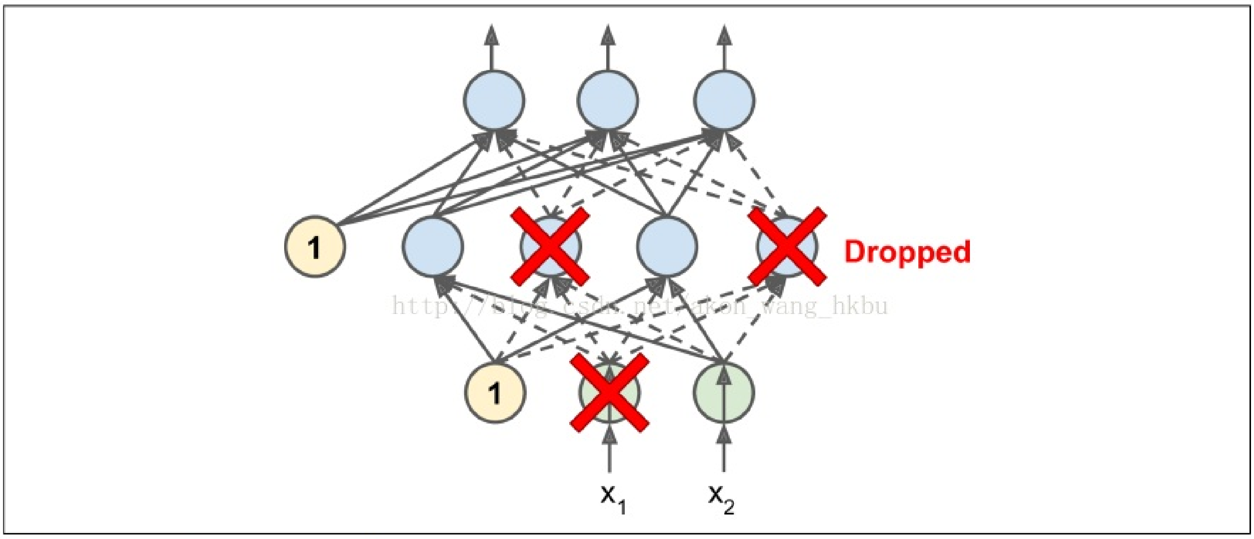


Figure 11-9. Dropout regularization

一开始这个技术是相当粗鲁，这是相当令人惊讶的。如果一个公司的员工每天早上被告知要掷硬币来决定是否上班，公司的表现会不会更好呢？那么，谁知道；也许会！公司显然将被迫适应这样的组织构架；它不能依靠任何一个人填写咖啡机或执行任何其他关键任务，所以这个专业知识将不得不分散在几个人身上。员工必须学会与其他的许多同事合作，而不仅仅是其中的一小部分。该公司将变得更有弹性。如果一个人离开了，并没有什么区别。目前还不清楚这个想法是否真的可以在公司实行，但它确实对于神经网络是可以的。神经元被dropout训练不能与其相邻的神经元共适应；他们必须尽可能让自己变得有用。他们也不能过分依赖一些输入神经元;他们必须注意他们的每个输入神经元。他们最终对输入的微小变化会不太敏感。最后，你会得到一个更强大的网络，更好地推广。

了解 dropout 的另一种方法是认识到每个训练步骤都会产生一个独特的神经网络。 由于每个神经元可以存在或不存在，总共有 `2 ^ N` 个可能的网络（其中 N 是可丢弃神经元的总数）。 这是一个巨大的数字，实际上不可能对同一个神经网络进行两次采样。 一旦你运行了 10,000 个训练步骤，你基本上已经训练了 10,000 个不同的神经网络（每个神经网络只有一个训练实例）。 这些神经网络显然不是独立的，因为它们共享许多权重，但是它们都是不同的。 由此产生的神经网络可以看作是所有这些较小的神经网络的平均集成。

有一个小而重要的技术细节。 假设 `p = 50%` ，在这种情况下，在测试期间，在训练期间神经元将被连接到两倍于（平均）的输入神经元。 为了弥补这个事实，我们需要在训练之后将每个神经元的输入连接权重乘以 0.5。 如果我们不这样做，每个神经元的总输入信号大概是网络训练的两倍，这不太可能表现良好。 更一般地说，我们需要将每个输入连接权重乘以训练后的保持概率（ `1-p` ）。 或者，我们可以在训练过程中将每个神经元的输出除以保持概率（这些替代方案并不完全等价，但它们工作得同样好）。

要使用 TensorFlow 实现dropout，可以简单地将 `dropout()` 函数应用于输入层和每个隐藏层的输出。 在训练过程中，这个功能随机丢弃一些节点（将它们设置为 0），并用保留概率来划分剩余项目。 训练结束后，这个函数什么都不做。下面的代码将dropout正则化应用于我们的三层神经网络：

注意：本书使用`tf.contrib.layers.dropout()`而不是`tf.layers.dropout()`（本章写作时不存在）。现在最好使用`tf.layers.dropout()`，因为`contrib`模块中的任何内容都可能会改变或被删除，恕不另行通知。`tf.layers.dropout()`函数几乎与`tf.contrib.layers.dropout()`函数相同，只是有一些细微差别。最重要的是：

- 您必须指定丢失率（率）而不是保持概率（`keep_prob`），其中`rate`简单地等于`1 - keep_prob`
- `is_training`参数被重命名为`training`。

```
X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")
```

```
training = tf.placeholder_with_default(False, shape=(), name='training')

dropout_rate = 0.5  # == 1 - keep_prob
X_drop = tf.layers.dropout(X, dropout_rate, training=training)

with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X_drop, n_hidden1, activation=tf.nn.relu,
                              name="hidden1")
    hidden1_drop = tf.layers.dropout(hidden1, dropout_rate, training=training)
    hidden2 = tf.layers.dense(hidden1_drop, n_hidden2, activation=tf.nn.relu,
                              name="hidden2")
    hidden2_drop = tf.layers.dropout(hidden2, dropout_rate, training=training)
    logits = tf.layers.dense(hidden2_drop, n_outputs, name="outputs")
```

```
with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")

with tf.name_scope("train"):
    optimizer = tf.train.MomentumOptimizer(learning_rate, momentum=0.9)
    training_op = optimizer.minimize(loss)

with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

```
n_epochs = 20
batch_size = 50

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={training: True, X: X_batch, y: y_batch})
        acc_test = accuracy.eval(feed_dict={X: mnist.test.images, y: mnist.test.labels
})
        print(epoch, "Test accuracy:", acc_test)

    save_path = saver.save(sess, "./my_model_final.ckpt")
```

你想在 `tensorflow.contrib.layers` 中使用 `dropout()` 函数，而不是 `tensorflow.nn` 中的那个。第一个在不训练的时候关掉（没有操作），这是你想要的，而第二个不是。

如果观察到模型过拟合，则可以增加 dropout 率（即，减少 `keep_prob` 超参数）。 相反，如果模型欠拟合训练集，则应尝试降低 dropout 率（即增加 `keep_prob` ）。 它也可以帮助增加大层的 dropout 率，并减少小层的 dropout 率。

dropout 似乎减缓了收敛速度，但通常会在调整得当时使模型更好。 所以，这通常值得花费额外的时间和精力。

Dropconnect是dropout的变体，其中单个连接随机丢弃而不是整个神经元。 一般而言，dropout表现会更好。

最大范数正则化

另一种在神经网络中非常流行的正则化技术被称为最大范数正则化：对于每个神经元，它约束输入连接的权重 `w`，使得 $\| \mathbf{w} \|_2 \leq r$，其中 `r` 是最大范数超参数，$\| \cdot \|_2$ 是 l2 范数。

我们通常通过在每个训练步骤之后计算 $\|\mathbf{w}\|_2$ 来实现这个约束，并且如果需要的话可以剪切 `w` $(\mathbf{w} \leftarrow \mathbf{w}\frac{r}{\| \mathbf{w} \|_2})$ 。

减少 `r` 增加了正则化的数量，并有助于减少过拟合。 最大范数正则化还可以帮助减轻梯度消失/爆炸问题（如果您不使用批量标准化）。

让我们回到 MNIST 的简单而简单的神经网络，只有两个隐藏层：

```
n_inputs = 28 * 28
n_hidden1 = 300
n_hidden2 = 50
n_outputs = 10

learning_rate = 0.01
momentum = 0.9

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")

with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu, name="hidden1")
    hidden2 = tf.layers.dense(hidden1, n_hidden2, activation=tf.nn.relu, name="hidden2"
)
    logits = tf.layers.dense(hidden2, n_outputs, name="outputs")

with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")

with tf.name_scope("train"):
    optimizer = tf.train.MomentumOptimizer(learning_rate, momentum)
    training_op = optimizer.minimize(loss)

with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

接下来，让我们来处理第一个隐藏层的权重，并创建一个操作，使用 `clip_by_norm()` 函数计算剪切后的权重。 然后我们创建一个赋值操作来将权值赋给权值变量：

```
threshold = 1.0
weights = tf.get_default_graph().get_tensor_by_name("hidden1/kernel:0")
clipped_weights = tf.clip_by_norm(weights, clip_norm=threshold, axes=1)
clip_weights = tf.assign(weights, clipped_weights)
```

我们也可以为第二个隐藏层做到这一点：

```
weights2 = tf.get_default_graph().get_tensor_by_name("hidden2/kernel:0")
clipped_weights2 = tf.clip_by_norm(weights2, clip_norm=threshold, axes=1)
clip_weights2 = tf.assign(weights2, clipped_weights2)
```

让我们添加一个初始化器和一个保存器：

```
init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

现在我们可以训练模型。 与往常一样，除了在运行 `training_op` 之后，我们运行 `clip_weights` 和 `clip_weights2` 操作：

```
n_epochs = 20
batch_size = 50
```

```
with tf.Session() as sess:                                              # not shown in
 the book
    init.run()                                                          # not shown
    for epoch in range(n_epochs):                                       # not shown
        for iteration in range(mnist.train.num_examples // batch_size):  # not shown
            X_batch, y_batch = mnist.train.next_batch(batch_size)       # not shown
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
            clip_weights.eval()
            clip_weights2.eval()                                        # not shown
        acc_test = accuracy.eval(feed_dict={X: mnist.test.images,       # not shown
                                            y: mnist.test.labels})      # not shown
        print(epoch, "Test accuracy:", acc_test)                        # not shown

    save_path = saver.save(sess, "./my_model_final.ckpt")               # not shown
```

上面的实现很简单，工作正常，但有点麻烦。 更好的方法是定义一个 `max_norm_regularizer()` 函数：

```
def max_norm_regularizer(threshold, axes=1, name="max_norm",
                         collection="max_norm"):
    def max_norm(weights):
        clipped = tf.clip_by_norm(weights, clip_norm=threshold, axes=axes)
        clip_weights = tf.assign(weights, clipped, name=name)
        tf.add_to_collection(collection, clip_weights)
        return None # there is no regularization loss term
    return max_norm
```

然后你可以调用这个函数来得到一个最大范数调节器（与你想要的阈值）。 当你创建一个隐藏层时，你可以将这个正则化器传递给 `kernel_regularizer` 参数：

```
n_inputs = 28 * 28
n_hidden1 = 300
n_hidden2 = 50
n_outputs = 10

learning_rate = 0.01
momentum = 0.9

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")
```

```
max_norm_reg = max_norm_regularizer(threshold=1.0)
with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu,
                              kernel_regularizer=max_norm_reg, name="hidden1")
    hidden2 = tf.layers.dense(hidden1, n_hidden2, activation=tf.nn.relu,
                              kernel_regularizer=max_norm_reg, name="hidden2")
    logits = tf.layers.dense(hidden2, n_outputs, name="outputs")
```

```
with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")

with tf.name_scope("train"):
    optimizer = tf.train.MomentumOptimizer(learning_rate, momentum)
    training_op = optimizer.minimize(loss)

with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

训练与往常一样，除了每次训练后必须运行重量裁剪操作：

请注意，最大范数正则化不需要在整体损失函数中添加正则化损失项，所以 `max_norm()` 函数返回 `None` 。 但是，在每个训练步骤之后，仍需要运行 `clip_weights` 操作，因此您需要能够掌握它。 这就是为什么 `max_norm()` 函数将 `clip_weights` 节点添加到最大范数剪裁操作的集合中的原因。您需要获取这些裁剪操作并在每个训练步骤后运行它们：

```
n_epochs = 20
batch_size = 50
```

```
clip_all_weights = tf.get_collection("max_norm")

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
            sess.run(clip_all_weights)
        acc_test = accuracy.eval(feed_dict={X: mnist.test.images,     # not shown in t
he book
                                            y: mnist.test.labels})    # not shown
        print(epoch, "Test accuracy:", acc_test)                      # not shown

    save_path = saver.save(sess, "./my_model_final.ckpt")             # not shown
```

数据增强

最后一个正则化技术，数据增强，包括从现有的训练实例中产生新的训练实例，人为地增加了训练集的大小。 这将减少过拟合，使之成为正则化技术。 诀窍是生成逼真的训练实例; 理想情况下，一个人不应该能够分辨出哪些是生成的，哪些不是生成的。 而且，简单地加白噪声也无济于事。 你应用的修改应该是可以学习的（白噪声不是）。

例如，如果您的模型是为了分类蘑菇图片，您可以稍微移动，旋转和调整训练集中的每个图片的大小，并将结果图片添加到训练集（见图 11-10）。 这迫使模型更能容忍图片中蘑菇的位置，方向和大小。 如果您希望模型对光照条件更加宽容，则可以类似地生成具有各种对比

度的许多图像。 假设蘑菇是对称的，你也可以水平翻转图片。 通过结合这些转换，可以大大增加训练集的大小。

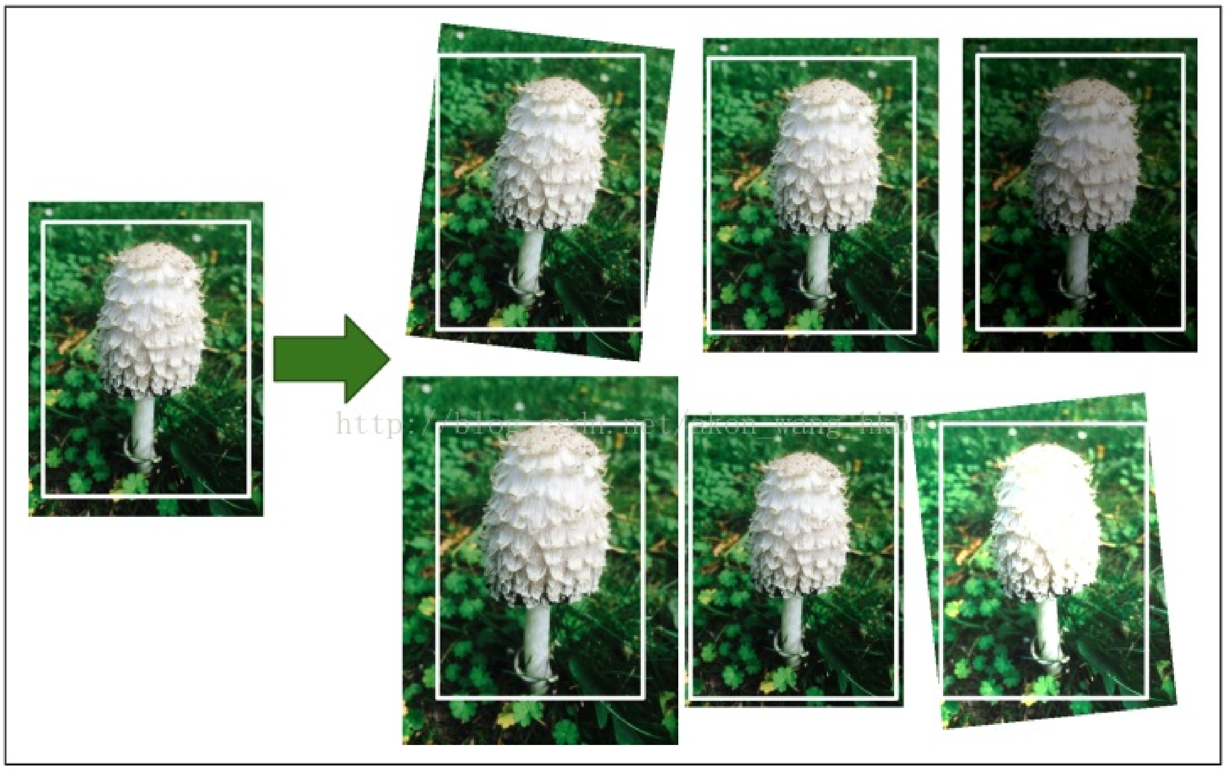


Figure 11-10. Generating new training instances from existing ones

通常最好在训练期间生成训练实例，而不是浪费存储空间和网络带宽。TensorFlow 提供了多种图像处理操作，例如转置（移位），旋转，调整大小，翻转和裁剪，以及调整亮度，对比度，饱和度和色调（请参阅 API 文档以获取更多详细信息）。 这可以很容易地为图像数据集实现数据增强。

训练非常深的神经网络的另一个强大的技术是添加跳过连接（跳过连接是将层的输入添加到更高层的输出时）。 我们将在第 13 章中谈论深度残差网络时探讨这个想法。

实践指南

在本章中，我们已经涵盖了很多技术，你可能想知道应该使用哪些技术。 表 11-2 中的配置在大多数情况下都能正常工作。

Table 11-2. Default DNN configuration

Initialization	He initialization
Activation function	ELU
Normalization	Batch Normalization
Regularization	Dropout
Optimizer	Adam
Learning rate schedule	None

当然，如果你能找到解决类似问题的方法，你应该尝试重用预训练的神经网络的一部分。

这个默认配置可能需要调整：

- 如果你找不到一个好的学习率（收敛速度太慢，所以你增加了训练速度，现在收敛速度很快，但是网络的准确性不是最理想的），那么你可以尝试添加一个学习率调整，如指数衰减。
- 如果你的训练集太小，你可以实现数据增强。
- 如果你需要一个稀疏的模型，你可以添加 l1 正则化混合（并可以选择在训练后将微小的权重归零）。如果您需要更稀疏的模型，您可以尝试使用 FTRL 而不是 Adam 优化以及 l1 正则化。
- 如果在运行时需要快速模型，则可能需要删除批量标准化，并可能用 leakyReLU 替换 ELU 激活函数。有一个稀疏的模型也将有所帮助。

有了这些指导方针，你现在已经准备好训练非常深的网络 - 好吧，如果你非常有耐心的话，那就是！如果使用单台机器，则可能需要等待几天甚至几个月才能完成训练。在下一章中，我们将讨论如何使用分布式 TensorFlow 在许多服务器和 GPU 上训练和运行模型。

练习

1. 使用 He 初始化随机选择权重，是否可以将所有权重初始化为相同的值？
2. 可以将偏置初始化为 0 吗？
3. 说出 ELU 激活功能与 ReLU 相比的三个优点。
4. 在哪些情况下，您想要使用以下每个激活函数：ELU，leaky ReLU（及其变体），ReLU，tanh，logistic 以及 softmax？
5. 使用 `MomentumOptimizer` 时，如果将 `momentum` 超参数设置得太接近 1（例如，0.99999），会发生什么情况？
6. 请列举您可以生成稀疏模型的三种方法。

7. dropout 是否会减慢训练？ 它是否会减慢推断（即预测新的实例）？
8. 深度学习。
   i. 建立一个 DNN，有五个隐藏层，每层 100 个神经元，使用 He 初始化和 ELU 激活函数。
   ii. 使用 Adam 优化和提前停止，请尝试在 MNIST 上进行训练，但只能使用数字 0 到 4，因为我们将在下一个练习中在数字 5 到 9 上进行迁移学习。 您需要一个包含五个神经元的 softmax 输出层，并且一如既往地确保定期保存检查点，并保存最终模型，以便稍后再使用它。
   iii. 使用交叉验证调整超参数，并查看你能达到什么准确度。
   iv. 现在尝试添加批量标准化并比较学习曲线：它是否比以前收敛得更快？ 它是否会产生更好的模型？
   v. 模型是否过拟合训练集？ 尝试将 dropout 添加到每一层，然后重试。 它有帮助吗？
9. 迁移学习。
   i. 创建一个新的 DNN，它复制先前模型的所有预训练的隐藏层，冻结它们，并用新的一层替换 softmax 输出层。
   ii. 在数字 5 到 9 训练这个新的 DNN ，每个数字只使用 100 个图像，需要多长时间？尽管样本这么少，你能达到高准确度吗？
   iii. 尝试缓存冻结的层，并再次训练模型：现在速度有多快？
   iv. 再次尝试复用四个隐藏层而不是五个。 你能达到更高的准确度吗？
   v. 现在，解冻前两个隐藏层并继续训练：您可以让模型表现得更好吗？
10. 辅助任务的预训练。
   i. 在本练习中，你将构建一个 DNN，用于比较两个 MNIST 数字图像，并预测它们是否代表相同的数字。 然后，你将复用该网络的较低层，来使用非常少的训练数据来训练 MNIST 分类器。 首先构建两个 DNN（我们称之为 DNN A 和 B），它们与之前构建的 DNN 类似，但没有输出层：每个 DNN 应该有五个隐藏层，每个层包含 100 个神经元，使用 He 初始化和 ELU 激活函数。 接下来，在两个 DNN 上添加一个输出层。 你应该使用 TensorFlow 的 `concat()函数和` axis = 1`，将两个 DNN 的输出沿着横轴连接，然后将结果输入到输出层。 输出层应该包含单个神经元，使用 logistic 激活函数。
   ii. 将 MNIST 训练集分为两组：第一部分应包含 55,000个 图像，第二部分应包含 5000 个图像。 创建一个生成训练批次的函数，其中每个实例都是从第一部分中挑选的一对 MNIST 图像。 一半的训练实例应该是属于同一类的图像对，而另一半应该是来自不同类别的图像。 对于每一对，如果图像来自同一类，训练标签应该为 0；如果来自不同类，则标签应为 1。
   iii. 在这个训练集上训练 DNN。 对于每个图像对，你可以同时将第一张图像送入 DNN A，将第二张图像送入 DNN B。整个网络将逐渐学会判断两张图像是否属于同一类别。
   iv. 现在通过复用和冻结 DNN A 的隐藏层，并添加 1 0个神经元的 softmax 输出层来创建一个新的 DNN。 在第二部分上训练这个网络，看看你是否可以实现较好的表现，尽管每类只有 500 个图像。

这些问题的答案在附录A中。

第12章 设备和服务器上的分布式TensorFlow

在第 11 章，我们讨论了几种可以明显加速训练的技术：更好的权重初始化，批量标准化，复杂的优化器等等。 但是，即使采用了所有这些技术，在具有单个 CPU 的单台机器上训练大型神经网络可能需要几天甚至几周的时间。

在本章中，我们将看到如何使用 TensorFlow 在多个设备（CPU 和 GPU）上分配计算并将它们并行运行（参见图 12-1）。 首先，我们会先在一台机器上的多个设备上分配计算，然后在多台机器上的多个设备上分配计算。

与其他神经网络框架相比，TensorFlow 对分布式计算的支持是其主要亮点之一。 它使您可以完全控制如何跨设备和服务器分布（或复制）您的计算图，并且可以让您以灵活的方式并行和同步操作，以便您可以在各种并行方法之间进行选择。

我们来看一些最流行的方法来并行执行和训练一个神经网络，这让我们不再需要等待数周才能完成训练算法，而最终可能只会等待几个小时。 这不仅可以节省大量时间，还意味着您可以更轻松地尝试各种模型，并经常重新训练模型上的新数据。

还有其他很好的并行化例子，包括当我们在微调模型时可以探索更大的超参数空间，并有效地运行大规模神经网络。

但我们必须先学会走路才能跑步。 我们先从一台机器上的几个 GPU 上并行化简单图形开始。

一台机器上多设备

只需添加 GPU 卡到单个机器，您就可以获得主要的性能提升。 事实上，在很多情况下，这就足够了。 你根本不需要使用多台机器。 例如，通常在单台机器上使用 8 个 GPU，而不是在多台机器上使用 16 个 GPU（由于多机器设置中的网络通信带来的额外延迟），可以同样快地训练神经网络。

在本节中，我们将介绍如何设置您的环境，以便 TensorFlow 可以在一台机器上使用多个 GPU 卡。 然后，我们将看看如何在可用设备上进行分布操作，并且并行执行它们。

安装

为了在多个 GPU 卡上运行 TensorFlow，首先需要确保 GPU 卡具有 NVidia 计算能力（大于或等于3.0）。 这包括 Nvidia 的 Titan，Titan X，K20 和 K40（如果你拥有另一张卡，你可以在 https://developer.nvidia.com/cuda-gpus 查看它的兼容性）。

如果您不拥有任何 GPU 卡，则可以使用具有 GPU 功能的主机服务器，如 Amazon AWS。在 ŽigaAvsec 的博客文章中，提供了在 Amazon AWS GPU 实例上使用 Python 3.5 设置 TensorFlow 0.9 的详细说明。将它更新到最新版本的 TensorFlow 应该不会太难。 Google 还发布了一项名为 Cloud Machine Learning 的云服务来运行 TensorFlow 图表。 2016 年 5 月，他们宣布他们的平台现在包括配备张量处理器（TPU）的服务器，专门用于机器学习的处理器，比许多 GPU 处理 ML 任务要快得多。 当然，另一种选择只是购买你自己的 GPU 卡。 Tim Dettmers 写了一篇很棒的博客文章来帮助你选择，他会定期更新它。

您必须下载并安装相应版本的 CUDA 和 cuDNN 库（如果您使用的是 TensorFlow 1.0.0，则为 CUDA 8.0 和 cuDNN 5.1），并设置一些环境变量，以便 TensorFlow 知道在哪里可以找到 CUDA 和 cuDNN。 详细的安装说明可能会相当迅速地更改，因此最好按照 TensorFlow 网站上的说明进行操作。

Nvidia 的 CUDA 允许开发者使用支持 CUDA 的 GPU 进行各种计算（不仅仅是图形加速）。 Nvidia 的 CUDA 深度神经网络库（cuDNN）是针对 DNN 的 GPU 加速原语库。 它提供了常用 DNN 计算的优化实现，例如激活层，归一化，前向和后向卷积以及池化（参见第 13 章）。 它是 Nvidia Deep Learning SDK 的一部分（请注意，它需要创建一个 Nvidia 开发者帐户才能下载它）。 TensorFlow 使用 CUDA 和 cuDNN 来控制 GPU 卡并加速计算（见图 12-2）。

您可以使用 `nvidia-smi` 命令来检查 CUDA 是否已正确安装。 它列出了可用的 GPU 卡以及每张卡上运行的进程：

最后，您必须安装支持 GPU 的 TensorFlow。 如果你使用 `virtualenv` 创建了一个独立的环境，你首先需要激活它：

```
$ cd $ML_PATH
# Your ML working directory (e.g., HOME/ml)
$ source env/bin/activate
```

然后安装合适的支持 GPU 的 TensorFlow 版本：

```
$ pip3 install --upgrade tensorflow-gpu
```

现在您可以打开一个 Python shell 并通过导入 TensorFlow 并创建一个会话来检查 TensorFlow 是否正确检测并使用 CUDA 和 cuDNN：

```
>>> import tensorflow as tf
I [...]/dso_loader.cc:108] successfully opened CUDA library libcublas.so locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcudnn.so locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcufft.so locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcuda.so.1 locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcurand.so locally
```

```
>>> sess = tf.Session()
[...]
I [...]/gpu_init.cc:102] Found device 0 with properties:
name: GRID K520
major: 3 minor: 0 memoryClockRate (GHz) 0.797
pciBusID 0000:00:03.0
Total memory: 4.00GiB
Free memory: 3.95GiB
I [...]/gpu_init.cc:126] DMA: 0
I [...]/gpu_init.cc:136] 0:   Y
I [...]/gpu_device.cc:839] Creating TensorFlow device
(/gpu:0) -> (device: 0, name: GRID K520, pci bus id: 0000:00:03.0)
```

看起来不错！TensorFlow 检测到 CUDA 和 cuDNN 库，并使用 CUDA 库来检测 GPU 卡（在这种情况下是 Nvidia Grid K520 卡）。

管理 GPU 内存

默认情况下，TensorFlow 会在您第一次运行图形时自动获取所有可用 GPU 中的所有 RAM，因此当第一个程序仍在运行时，您将无法启动第二个 TensorFlow 程序。 如果你尝试，你会得到以下错误：

```
E [...]/cuda_driver.cc:965] failed to allocate 3.66G (3928915968 bytes) from device: C
UDA_ERROR_OUT_OF_MEMORY
```

一种解决方案是在不同的 GPU 卡上运行每个进程。 为此，最简单的选择是设置 `CUDA_VISIBLE_DEVICES` 环境变量，以便每个进程只能看到对应的 GPU 卡。 例如，你可以像这样启动两个程序：

```
$ CUDA_VISIBLE_DEVICES=0,1 python3 program_1.py
# and in another terminal:
$ CUDA_VISIBLE_DEVICES=3,2 python3 program_2.py
```

程序 #1 只会看到 GPU 卡 0 和 1（分别编号为 0 和 1），程序 #2 只会看到 GPU 卡 2 和 3（分别编号为 1 和 0）。 一切都会正常工作（见图 12-3）。

另一种选择是告诉 TensorFlow 只抓取一小部分内存。 例如，要使 TensorFlow 只占用每个 GPU 内存的 40%，您必须创建一个 `ConfigProto` 对象，将其 `gpu_options.per_process_gpu_memory_fraction` 选项设置为 0.4，并使用以下配置创

建 `session` ：

```
config = tf.ConfigProto()
config.gpu_options.per_process_gpu_memory_fraction = 0.4
session = tf.Session(config=config)
```

现在像这样的两个程序可以使用相同的 GPU 卡并行运行（但不是三个，因为 `3×0.4> 1` ）。见图 12-4。

如果在两个程序都运行时运行 `nvidia-smi` 命令，则应该看到每个进程占用每个卡的总 RAM 大约 40%：

另一种选择是告诉 TensorFlow 只在需要时才抓取内存。 为此，您必须将 `config.gpu_options.allow_growth` 设置为 `True` 。但是，TensorFlow 一旦抓取内存就不会释放内存（以避免内存碎片），因此您可能会在一段时间后内存不足。 是否使用此选项可能难以确定，因此一般而言，您可能想要坚持之前的某个选项。

好的，现在你已经有了一个支持 GPU 的 TensorFlow 安装。 让我们看看如何使用它！

设备布置操作

TensorFlow 白皮书介绍了一种友好的动态布置器算法，该算法能够自动将操作分布到所有可用设备上，并考虑到以前运行图中所测量的计算时间，估算每次操作的输入和输出张量的大小， 每个设备可用的 RAM，传输数据进出设备时的通信延迟，来自用户的提示和约束等等。不幸的是，这种复杂的算法是谷歌内部的，它并没有在 TensorFlow 的开源版本中发布。它被排除在外的原因似乎是，由用户指定的一小部分放置规则实际上比动态放置器放置的更有效。 然而，TensorFlow 团队正在努力改进它，并且最终可能会被开放。

在此之前，TensorFlow都是简单的放置，它（如其名称所示）非常基本。

简单放置

无论何时运行图形，如果 TensorFlow 需要求值尚未放置在设备上的节点，则它会使用简单放置器将其放置在未放置的所有其他节点上。 简单放置尊重以下规则：

- 如果某个节点已经放置在图形的上一次运行中的某个设备上，则该节点将保留在该设备上。
- 否则，如果用户将一个节点固定到设备上（下面介绍），则放置器将其放置在该设备上。
- 否则，它默认为 GPU#0，如果没有 GPU，则默认为 CPU。

正如您所看到的，将操作放在适当的设备上主要取决于您。 如果您不做任何事情，整个图表将被放置在默认设备上。 要将节点固定到设备上，您必须使用 `device()` 函数创建一个设备块。 例如，以下代码将变量 `a` 和常量 `b` 固定在 CPU 上，但乘法节点 `c` 不固定在任何设备上，因此将放置在默认设备上：

```
with tf.device("/cpu:0"):
    a = tf.Variable(3.0)
    b = tf.constant(4.0)

c = a * b
```

其中， `"/cpu:0"` 设备合计多 CPU 系统上的所有 CPU。 目前没有办法在特定 CPU 上固定节点或仅使用所有 CPU 的子集。

记录放置位置

让我们检查一下简单的放置器是否遵守我们刚刚定义的布局约束条件。 为此，您可以将 `log_device_placement` 选项设置为 `True` ；这告诉放置器在放置节点时记录消息。例如：

```
>>> config = tf.ConfigProto()
>>> config.log_device_placement = True
>>> sess = tf.Session(config=config)
I [...] Creating TensorFlow device (/gpu:0) -> (device: 0, name: GRID K520, pci bus id
: 0000:00:03.0)
[...]
>>> x.initializer.run(session=sess)
I [...] a: /job:localhost/replica:0/task:0/cpu:0
I [...] a/read: /job:localhost/replica:0/task:0/cpu:0
I [...] mul: /job:localhost/replica:0/task:0/gpu:0
I [...] a/Assign: /job:localhost/replica:0/task:0/cpu:0
I [...] b: /job:localhost/replica:0/task:0/cpu:0
I [...] a/initial_value: /job:localhost/replica:0/task:0/cpu:0
>>> sess.run(c)
12
```

`Info` 中以大写字母 `I` 开头的行是日志消息。 当我们创建一个会话时，TensorFlow 会记录一条消息，告诉我们它已经找到了一个 GPU 卡（在这个例子中是 Grid K520 卡）。 然后，我们第一次运行图形（在这种情况下，当初始化变量 `a` 时），简单布局器运行，并将每个节点放置在分配给它的设备上。正如预期的那样，日志消息显示所有节点都放在 `"/cpu:0"` 上，除了乘法节点，它以默认设备 `"/gpu:0"` 结束（您可以先忽略前缀： `/job:localhost/replica:0/task:0;` 我们将在一会儿讨论它）。 注意，我们第二次运行图（计算 `c` ）时，由于 TensorFlow 需要计算的所有节点 `c` 都已经放置，所以不使用布局器。

动态放置功能

创建设备块时，可以指定一个函数，而不是设备名称。TensorFlow 会调用这个函数来进行每个需要放置在设备块中的操作，并且该函数必须返回设备的名称来固定操作。 例如，以下代码将固定所有变量节点到 `"/cpu:0"` （在本例中只是变量 `a` ）和所有其他节点到 `"/gpu:0"` ：

```
def variables_on_cpu(op):
    if op.type == "Variable":
        return "/cpu:0"
    else:
        return "/gpu:0"

with tf.device(variables_on_cpu):
    a = tf.Variable(3.0)
    b = tf.constant(4.0)
    c = a * b
```

您可以轻松实现更复杂的算法，例如以循环方式用GPU锁定变量。

操作和内核

对于在设备上运行的 TensorFlow 操作，它需要具有该设备的实现；这被称为内核。 许多操作对于 CPU 和 GPU 都有内核，但并非全部都是。 例如，TensorFlow 没有用于整数变量的 GPU 内核，因此当 TensorFlow 尝试将变量i放置到 GPU#0 时，以下代码将失败：

```
>>> with tf.device("/gpu:0"):
...     i = tf.Variable(3)
[...]
>>> sess.run(i.initializer)
Traceback (most recent call last):
[...]
tensorflow.python.framework.errors.InvalidArgumentError: Cannot assign a device to nod
e 'Variable': Could not satisfy explicit device specification
```

请注意，TensorFlow 推断变量必须是 `int32` 类型，因为初始化值是一个整数。 如果将初始化值更改为 3.0 而不是 3，或者如果在创建变量时显式设置 `dtype = tf.float32` ，则一切正常。

软放置

默认情况下，如果您尝试在操作没有内核的设备上固定操作，则当 TensorFlow 尝试将操作放置在设备上时，您会看到前面显示的异常。 如果您更喜欢 TensorFlow 回退到 CPU，则可以将 `allow_soft_placement` 配置选项设置为 `True` ：

```
with tf.device("/gpu:0"):
    i = tf.Variable(3)

config = tf.ConfigProto()
config.allow_soft_placement = True
sess = tf.Session(config=config)
sess.run(i.initializer)  # the placer runs and falls back to /cpu:0
```

到目前为止，我们已经讨论了如何在不同设备上放置节点。 现在让我们看看 TensorFlow 如何并行运行这些节点。

并行运行

当 TensorFlow 运行图时，它首先找出需要求值的节点列表，然后计算每个节点有多少依赖关系。 然后 TensorFlow 开始求值具有零依赖关系的节点（即源节点）。 如果这些节点被放置在不同的设备上，它们显然会被并行求值。 如果它们放在同一个设备上，它们将在不同的线程中进行求值，因此它们也可以并行运行（在单独的 GPU 线程或 CPU 内核中）。

TensorFlow 管理每个设备上的线程池以并行化操作（参见图 12-5）。 这些被称为 inter-op 线程池。 有些操作具有多线程内核：它们可以使用其他线程池（每个设备一个）称为 intra-op 线程池（下面写成内部线程池）。

例如，在图 12-5 中，操作 `A` ， `B` 和 `C` 是源操作，因此可以立即进行求值。 操作 `A` 和 `B` 放置在 GPU#0 上，因此它们被发送到该设备的内部线程池，并立即进行并行求值。 操作A正好有一个多线程内核; 它的计算被分成三部分，这些部分由内部线程池并行执行。 操作 `C` 转到 GPU#1 的内部线程池。

一旦操作 `C` 完成，操作 `D` 和 `E` 的依赖性计数器将递减并且都将达到 0，因此这两个操作将被发送到操作内线程池以执行。

您可以通过设置 `inter_op_parallelism_threads` 选项来控制内部线程池的线程数。 请注意，您开始的第一个会话将创建内部线程池。 除非您将 `use_per_session_threads` 选项设置为 `True` ，否则所有其他会话都将重用它们。 您可以通过设置 `intra_op_parallelism_threads` 选项来控制每个内部线程池的线程数。

控制依赖关系

在某些情况下，即使所有依赖的操作都已执行，推迟对操作的求值可能也是明智之举。例如，如果它使用大量内存，但在图形中只需要更多内存，则最好在最后一刻对其进行求值，以避免不必要地占用其他操作可能需要的 RAM。 另一个例子是依赖位于设备外部的数据的一组操作。 如果它们全部同时运行，它们可能会使设备的通信带宽达到饱和，并最终导致所有等待 I/O。 其他需要传递数据的操作也将被阻止。 顺序执行这些通信繁重的操作将是比较好的，这样允许设备并行执行其他操作。

推迟对某些节点的求值，一个简单的解决方案是添加控制依赖关系。 例如，下面的代码告诉 TensorFlow 仅在求值完 `a` 和 `b` 之后才求值 `x` 和 `y` ：

```
a = tf.constant(1.0)
b = a + 2.0

with tf.control_dependencies([a, b]):
    x = tf.constant(3.0)
    y = tf.constant(4.0)

z = x + y
```

显然，由于 `z` 依赖于 `x` 和 `y` ，所以求值 `z` 也意味着等待 `a` 和 `b` 进行求值，即使它并未显式存在于 `control_dependencies()` 块中。 此外，由于 `b` 依赖于 `a` ，所以我们可以通过在 `[b]` 而不是 `[a,b]` 上创建控制依赖关系来简化前面的代码，但在某些情况下，“显式比隐式更好”。

很好！现在你知道了：

- 如何以任何您喜欢的方式在多个设备上进行操作
- 这些操作如何并行执行
- 如何创建控制依赖性来优化并行执行

是时候将计算分布在多个服务器上了！

多个服务器的多个设备

要跨多台服务器运行图形，首先需要定义一个集群。 一个集群由一个或多个 TensorFlow 服务器组成，称为任务，通常分布在多台机器上（见图 12-6）。 每项任务都属于一项作业。 作业只是一组通常具有共同作用的任务，例如跟踪模型参数（例如，参数服务器通常命名为 `"ps"` ，parameter server）或执行计算（这样的作业通常被命名为 `"worker"` ）。

以下集群规范定义了两个作业 `"ps"` 和 `"worker"` ，分别包含一个任务和两个任务。 在这个例子中，机器A托管着两个 TensorFlow 服务器（即任务），监听不同的端口：一个是 `"ps"` 作业的一部分，另一个是 `"worker"` 作业的一部分。 机器B仅托管一台 TensorFlow 服务器，这是 `"worker"` 作业的一部分。

```
cluster_spec = tf.train.ClusterSpec({
    "ps": [
        "machine-a.example.com:2221",  # /job:ps/task:0
        ],
    "worker": [
        "machine-a.example.com:2222",  # /job:worker/task:0
        "machine-b.example.com:2222",  # /job:worker/task:1
        ]})
```

要启动 TensorFlow 服务器，您必须创建一个服务器对象，并向其传递集群规范（以便它可以与其他服务器通信）以及它自己的作业名称和任务编号。 例如，要启动第一个辅助任务，您需要在机器 A 上运行以下代码：

```
server = tf.train.Server(cluster_spec, job_name="worker", task_index=0)
```

每台机器只运行一个任务通常比较简单，但前面的例子表明 TensorFlow 允许您在同一台机器上运行多个任务（如果需要的话）。 如果您在一台机器上安装了多台服务器，则需要确保它们不会全部尝试抓取每个 GPU 的所有 RAM，如前所述。 例如，在图12-6中， `"ps"` 任务没

有看到 GPU 设备，想必其进程是使用 `CUDA_VISIBLE_DEVICES =""` 启动的。 请注意，CPU由位于同一台计算机上的所有任务共享。

如果您希望进程除了运行 TensorFlow 服务器之外什么都不做，您可以通过告诉它等待服务器使用 `join()` 方法来完成，从而阻塞主线程（否则服务器将在您的主线程退出）。 由于目前没有办法阻止服务器，这实际上会永远阻止：

```
server.join()  # blocks until the server stops (i.e., never)
```

开始一个会话

一旦所有任务启动并运行（但还什么都没做），您可以从位于任何机器上的任何进程（甚至是运行中的进程）中的客户机上的任何服务器上打开会话，并使用该会话像普通的本地会议一样。比如：

```
a = tf.constant(1.0)
b = a + 2
c = a * 3

with tf.Session("grpc://machine-b.example.com:2222") as sess:
    print(c.eval())  # 9.0
```

这个客户端代码首先创建一个简单的图形，然后在位于机器 B（我们称之为主机）上的 TensorFlow 服务器上打开一个会话，并指示它求值 `c`。 主设备首先将操作放在适当的设备上。 在这个例子中，因为我们没有在任何设备上进行任何操作，所以主设备只将它们全部放在它自己的默认设备上 - 在这种情况下是机器 B 的 GPU 设备。 然后它只是按照客户的指示求值 `c`，并返回结果。

主机和辅助服务

客户端使用 gRPC 协议（Google Remote Procedure Call）与服务器进行通信。 这是一个高效的开源框架，可以调用远程函数，并通过各种平台和语言获取它们的输出。它基于 HTTP2，打开一个连接并在整个会话期间保持打开状态，一旦建立连接就可以进行高效的双向通信。

数据以协议缓冲区的形式传输，这是另一种开源 Google 技术。 这是一种轻量级的二进制数据交换格式。

TensorFlow 集群中的所有服务器都可能与集群中的任何其他服务器通信，因此请确保在防火墙上打开适当的端口。

每台 TensorFlow 服务器都提供两种服务：主服务和辅助服务。 主服务允许客户打开会话并使用它们来运行图形。 它协调跨任务的计算，依靠辅助服务实际执行其他任务的计算并获得结果。

固定任务的操作

通过指定作业名称，任务索引，设备类型和设备索引，可以使用设备块来锁定由任何任务管理的任何设备上的操作。 例如，以下代码将 `a` 固定在 `"ps"` 作业（即机器 A 上的 CPU）中第一个任务的 CPU，并将 `b` 固定在 `"worker"` 作业的第一个任务管理的第二个 GPU （这是 A 机上的 GPU#1）。 最后， `c` 没有固定在任何设备上，所以主设备将它放在它自己的默认设备上（机器 B 的 GPU#0 设备）。

```
with tf.device("/job:ps/task:0/cpu:0")
    a = tf.constant(1.0)

with tf.device("/job:worker/task:0/gpu:1")
    b = a + 2

c = a + b
```

如前所述，如果您省略设备类型和索引，则 TensorFlow 将默认为该任务的默认设备; 例如，将操作固定到 `"/job:ps/task:0"` 会将其放置在 `"ps"` 作业（机器 A 的 CPU）的第一个任务的默认设备上。 如果您还省略了任务索引（例如， `"/job:ps"` ），则 TensorFlow 默认为 `"/task:0"` 。如果省略作业名称和任务索引，则 TensorFlow 默认为会话的主任务。

跨多个参数服务器的分片变量

正如我们很快会看到的那样，在分布式设置上训练神经网络时，常见模式是将模型参数存储在一组参数服务器上（即 `"ps"` 作业中的任务），而其他任务则集中在计算上（即， `"worker"` 工作中的任务）。 对于具有数百万参数的大型模型，在多个参数服务器上分割这些参数非常有用，可以降低饱和单个参数服务器网卡的风险。 如果您要将每个变量手动固定到不同的参数服务器，那将非常繁琐。 幸运的是，TensorFlow 提供了 `replica_device_setter()` 函数，它以循环方式在所有 `"ps"` 任务中分配变量。 例如，以下代码将五个变量引入两个参数服务器：

```
with tf.device(tf.train.replica_device_setter(ps_tasks=2):
    v1 = tf.Variable(1.0)  # pinned to /job:ps/task:0
    v2 = tf.Variable(2.0)  # pinned to /job:ps/task:1
    v3 = tf.Variable(3.0)  # pinned to /job:ps/task:0
    v4 = tf.Variable(4.0)  # pinned to /job:ps/task:1
    v5 = tf.Variable(5.0)  # pinned to /job:ps/task:0
```

您不必传递 `ps_tasks` 的数量，您可以传递集群 `spec = cluster_spec` ，TensorFlow 将简单计算 `"ps"` 作业中的任务数。

如果您在块中创建其他操作，则不仅仅是变量，TensorFlow 会自动将它们连接到 `"/job:worker"` ，默认为第一个由 `"worker"` 作业中第一个任务管理的设备。 您可以通过设置 `worker_device` 参数将它们固定到其他设备，但更好的方法是使用嵌入式设备块。 内部设备块可以覆盖在外部块中定义的作业，任务或设备。 例如：

```
with tf.device(tf.train.replica_device_setter(ps_tasks=2)):
    v1 = tf.Variable(1.0)  # pinned to /job:ps/task:0 (+ defaults to /cpu:0)
    v2 = tf.Variable(2.0)  # pinned to /job:ps/task:1 (+ defaults to /cpu:0)
    v3 = tf.Variable(3.0)  # pinned to /job:ps/task:0 (+ defaults to /cpu:0)
    [...]
    s = v1 + v2            # pinned to /job:worker (+ defaults to task:0/gpu:0)
    with tf.device("/gpu:1"):
        p1 = 2 * s         # pinned to /job:worker/gpu:1 (+ defaults to /task:0)

        with tf.device("/task:1"):
            p2 = 3 * s     # pinned to /job:worker/task:1/gpu:1
```

这个例子假设参数服务器是纯 CPU 的，这通常是这种情况，因为它们只需要存储和传送参数，而不是执行密集计算。

第13章 卷积神经网络

尽管 IBM 的深蓝超级计算机在1996年击败了国际象棋世界冠军 Garry Kasparvo，直到近几年计算机都不能可靠地完成一些看起来较为复杂的任务，比如判别照片中是否有狗以及识别语音。为什么这些任务对于人类而言如此简单？答案在于感知主要发生在我们意识领域之外，在我们大脑中的专门视觉，听觉和其他感官模块内。当感官信息达到我们的意识时，它已经被装饰了高级特征;例如，当你看着一只可爱的小狗的照片时，你不能选择不看这只小狗，或不注意它的可爱。你也不能解释你如何认出这是一只可爱的小狗，这对你来说很明显。因此，我们不能相信我们的主观经验：感知并不是微不足道的，理解它我们必须看看感官模块是如何工作的。

卷积神经网络（CNN）是从大脑视觉皮层的研究中出现的，自 20 世纪 80 年代以来它们一直用于图像识别。在过去的几年里，由于计算能力的增加，可用训练数据的数量以及第 11 章介绍的训练深度网络的技巧，CNN 致力于在某些复杂的视觉任务中做出超出人类的表现。他们使图像搜索服务，自动驾驶汽车，视频自动分类系统等变得强大。此外，CNN 并不局限于视觉感知：它们在其他任务中也很成功，如语音识别或自然语言处理（NLP）; 然而，我们现在将专注于视觉应用。

在本章中，我们将介绍 CNN 的来源，构建它们模块的外观以及如何使用 TensorFlow 实现它们。然后我们将介绍一些最好的 CNN 架构。

视觉皮层的结构

David H.Hubel 和 Torsten Wiesel 在 1958 年和 1959 年对猫进行了一系列实验（以及几年后在猴子上的实验），对视觉皮层的结构提供了重要的见解（1981 年作者因此获得了诺贝尔生理和医学奖）。具体来说，他们发现视皮层中的许多神经元有一个小的局部感受野，这意味着它们只对位于视野中有限的一部分区域的视觉刺激起作用（见图 13-1，五个神经元的局部感受野由虚线圆圈表示）。不同神经元的感受野可能重叠，并且它们一起平铺了整个视野。此外，作者表明，一些神经元只对水平线方向的图像作出反应，而另一些神经元只对不同方向的线作出反应（两个神经元可能具有相同的感受野，但对不同方向的线作出反应）。他们还注意到一些神经元具有较大的感受野，并且它们对较复杂的模式作出反应，这些模式是较低层模式的组合。这些观察结果让我们想到：更高级别的神经元是基于相邻低级神经元的输出（在图 13-1 中，请注意，每个神经元只与来自前一层的少数神经元相连）。这个强大的结构能够检测视野中任何区域的各种复杂图案。

这些对视觉皮层的研究启发了 1980 年推出的新认知机（neocognitron），后者逐渐演变为我们现在称之为卷积神经网络。一个重要的里程碑是 Yann LeCun，LéonBottou，Yoshua Bengio 和 Patrick Haffner 于 1998 年发表的一篇论文，该论文引入了着名的 LeNet-5 架构，

广泛用于识别手写支票号码。这个架构有一些你已经知道的构建块，比如完全连接层和 Sigmoid 激活函数，但是它还引入了两个新的构建块：卷积层和池化层。现在我们来看看他们。

卷积层

CNN 最重要的组成部分是卷积层：第一卷积层中的神经元不是连接到输入图像中的每一个像素（就像它们在前面的章节中那样），而是仅仅连接到它们的局部感受野中的像素（参见图 13-2）。 进而，第二卷积层中的每个神经元只与位于第一层中的小矩形内的神经元连接。 这种架构允许网络专注于第一隐藏层中的低级特征，然后将其组装成下一隐藏层中的高级特征，等等。 这种层次结构在现实世界的图像中是很常见的，这也是 CNN 在图像识别方面效果很好的原因之一。

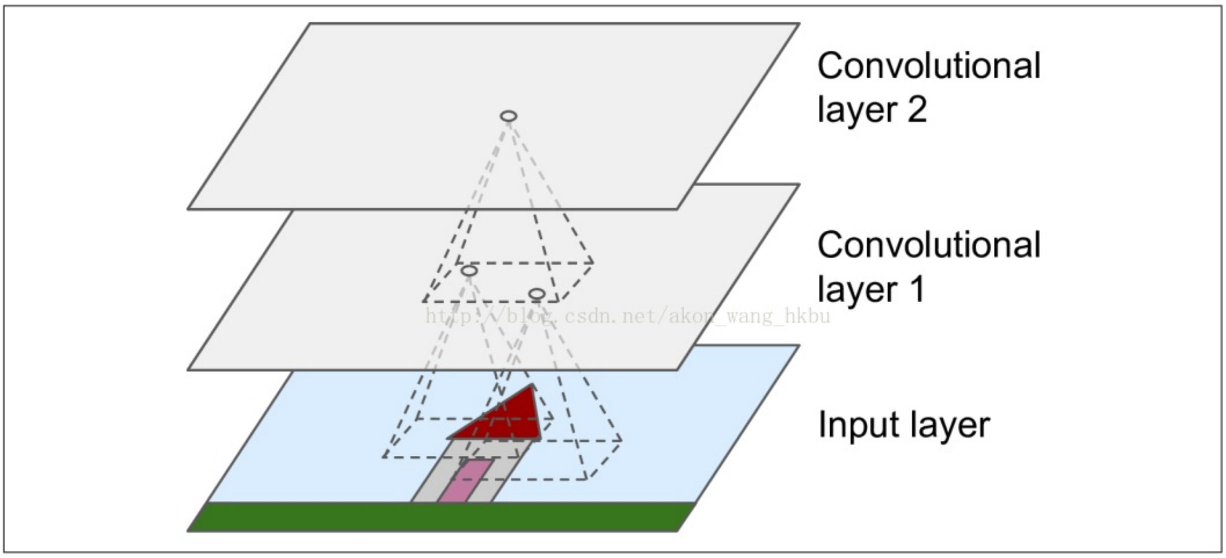


Figure 13-2. CNN layers with rectangular local receptive fields

到目前为止，我们所看到的所有多层神经网络都有由一长串神经元组成的层，在输入到神经网络之前我们必须将输入图像压缩成 **1D**。 现在，每个图层都以 **2D** 表示，这使得神经元与其相应的输入进行匹配变得更加容易。

位于给定层的第 `i` 行第 `j` 列的神经元连接到位于前一层中的神经元的输出的第i行到第 $i + f_h - 1$ 行，第 `j` 列到第 $j + f_w - 1$ 列。 `fh` 和 `fw` 是局部感受野的高度和宽度（见图 13-3）。 为了使图层具有与前一图层相同的高度和宽度，通常在输入周围添加零，如图所示。 这被称为零填充.

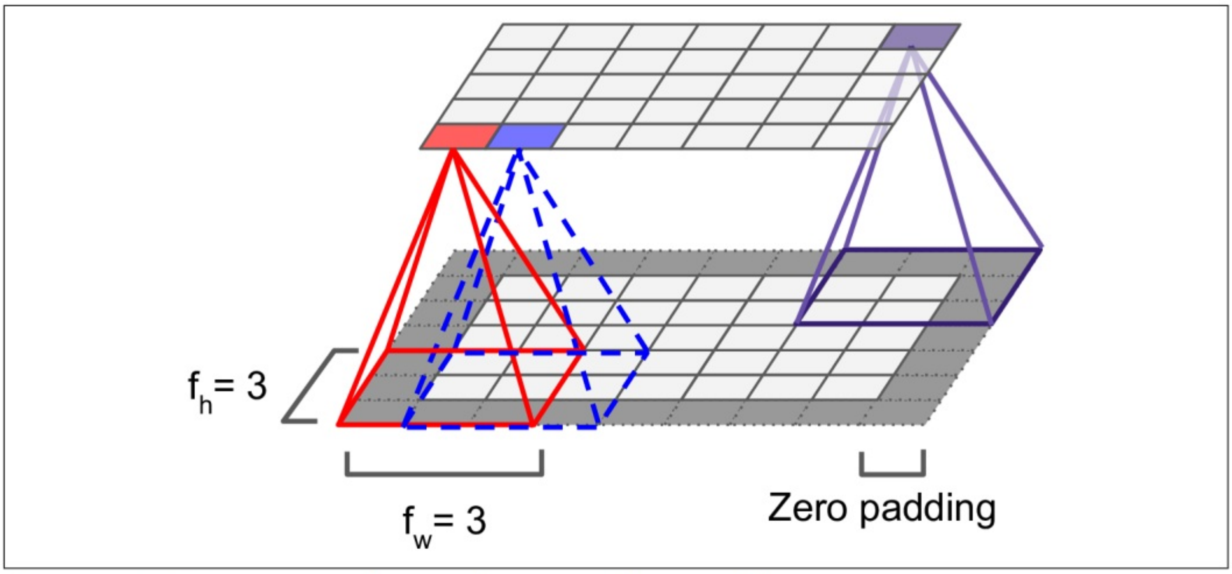


Figure 13-3. Connections between layers and zero padding

如图 13-4 所示，通过将局部感受野隔开，还可以将较大的输入层连接到更小的层。 两个连续的感受野之间的距离被称为步幅。 在图中，一个 `5×7` 的输入层（加零填充）连接到一个 `3×4` 层，使用 `3×3` 的卷积核和一个步幅为 2（在这个例子中，步幅在两个方向是相同的，但是它并不一定总是如此）。 位于上层第 `i` 行第 `j` 列的神经元与位于前一层中的神经元的输出连接的第 $i \times s_h$ 至 $i \times s_h + f_h - 1$ 行，第 $j \times s_w + f_w - 1$ 列，`Sh` 和 `Sw` 是垂直和水平的步幅。

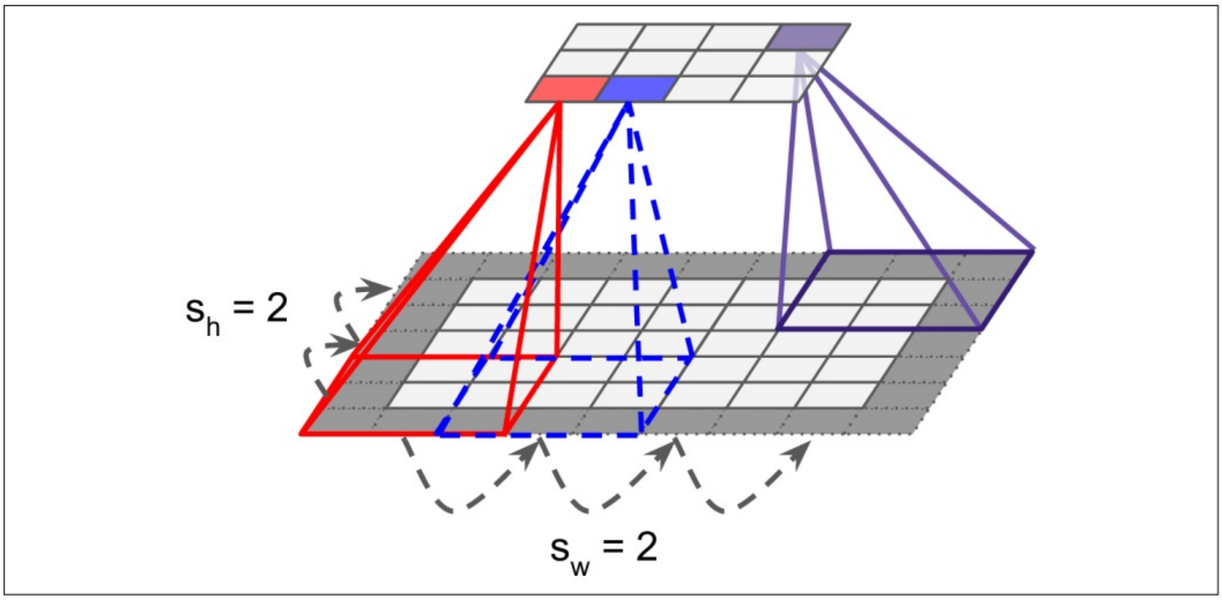


Figure 13-4. Reducing dimensionality using a stride

卷积核/过滤器

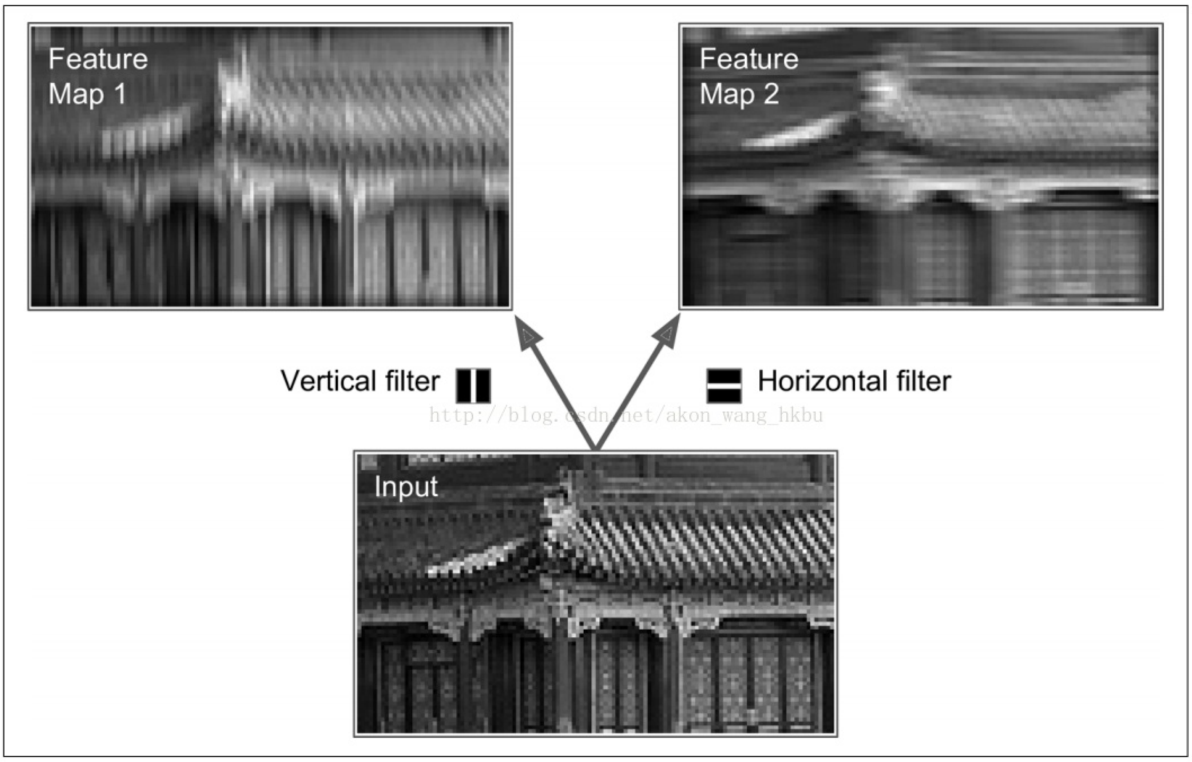


Figure 13-5. Applying two different filters to get two feature maps

神经元的权重可以表示为局部感受野大小的小图像。 例如，图 13-5 显示了两个可能的权重集，称为过滤器（或卷积核）。第一个表示为中间有一条垂直的白线的黑色正方形（除了中间一列外，这是一个充满 0 的 `7×7` 矩阵，除了中央垂直线是 1）。 使用这些权重的神经元会忽略除了中央垂直线以外感受野的一切（因为除位于中央垂直线以外，所有的输入都将乘 0）。第二个卷积核是一个黑色的正方形，中间有一条水平的白线。 再一次，使用这些权重的神经元将忽略除了中心水平线之外的局部感受野中的一切。

现在，如果一个图层中的所有神经元都使用相同的垂直线卷积核（以及相同的偏置项），并且将网络输入到图 13-5（底部图像）中所示的输入图像，则该图层将输出左上图像。 请注意，垂直的白线得到增强，其余的变得模糊。 类似地，如果所有的神经元都使用水平线卷积核，右上角的图像就是你所得到的。 注意到水平的白线得到增强，其余的则被模糊了。因此，使用相同卷积和的一个充满神经元的图层将为您提供一个特征映射，该特征映射突出显示图像中与卷积和最相似的区域。 在训练过程中，CNN 为其任务找到最有用的卷积和，并学习将它们组合成更复杂的模式（例如，交叉是图像中垂直卷积和和水平卷积和都激活的区域）。

叠加的多个特征映射

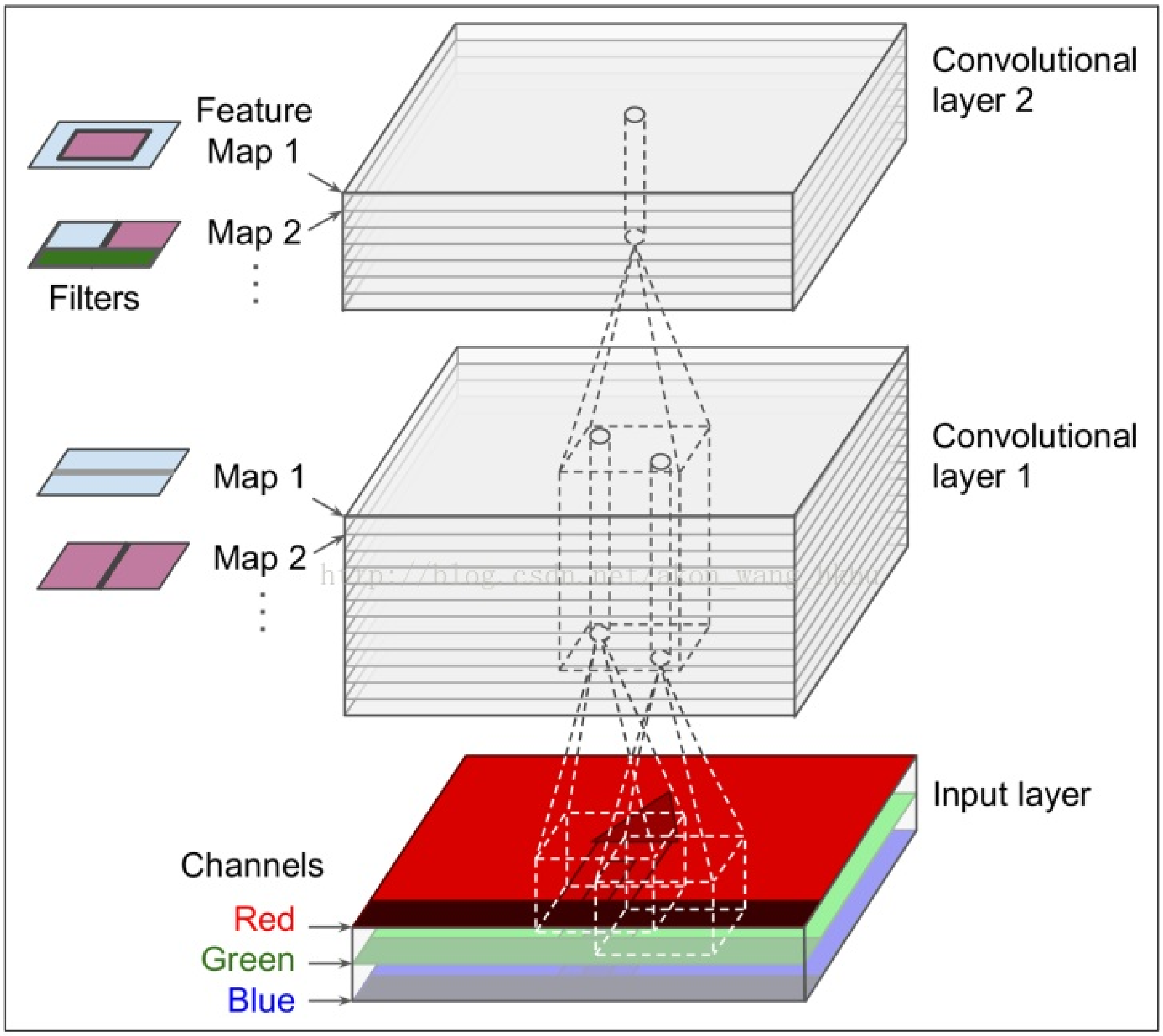


Figure 13-6. Convolution layers with multiple feature maps, and images with three channels

到目前为止，为了简单起见，我们已经将每个卷积层表示为一个薄的二维层，但是实际上它是由几个相同大小的特征映射组成的，所以使用3D图表示其会更加准确（见图 13-6）。 在一个特征映射中，所有神经元共享相同的参数（权重和偏置，权值共享），但是不同的特征映射可能具有不同的参数。 神经元的感受野与前面描述的相同，但是它延伸到所有先前的层的特征映射。 简而言之，卷积层同时对其输入应用多个卷积核，使其能够检测输入中的任何位置的多个特征。

事实上，特征地图中的所有神经元共享相同的参数会显着减少模型中的参数数量，但最重要的是，一旦 **CNN** 学会识别一个位置的模式，就可以在任何其他位置识别它。 相比之下，一旦一个常规 **DNN** 学会识别一个位置的模式，它只能在该特定位置识别它。

而且，输入图像也由多个子图层组成：每个颜色通道一个。 通常有三种：红色，绿色和蓝色（RGB）。 灰度图像只有一个通道，但是一些图像可能更多 - 例如捕捉额外光频（如红外线）的卫星图像。

具体地，位于给定卷积层L中的特征映射k的i行，j列中的神经元连接到前一层（L-1）位于 $i \times s_w$ to $i \times s_w + f_w - 1$ 行，$j \times s_h$ to $j \times s_h + f_h - 1$ 列的神经元的输出。请注意，位于同一行第 `i` 列和第 `j` 列但位于不同特征映射中的所有神经元都连接到上一层中完全相同神经元的输出。

公式 13-1 在一个总结前面解释的大的数学公式：它展示了如何计算卷积层中给定神经元的输出。它是计算所有投入的加权总并且加上偏置。

Equation 13-1. Computing the output of a neuron in a convolutional layer

$$z_{i,j,k} = b_k + \sum_{u=1}^{f_h} \sum_{v=1}^{f_w} \sum_{k'=1}^{f_{n'}} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{with} \begin{cases} i' = u \cdot s_h + f_h - 1 \\ j' = v \cdot s_w + f_w - 1 \end{cases}$$

- $z_{i,j,k}$ 是卷积层（L层）特征映射k中位于第 `i` 行第 `j` 列的神经元的输出.
- 如前所述，s_h 和 s_w 是垂直和水平的步幅，f_h 和 f_w 是感受野的高度和宽度，$f_{n'}$ 是前一层（第 `l - 1` 层）的特征映射的数量。
- $x_{i',j',k'}$ 是位于层 `L-1`，`i'` 行，`j'` 列，特征映射 `k'`（或者如果前一层是输入层的通道 `k'`）的神经元的输出。
- b_k 是特征映射 `k` 的偏置项（在 `L` 层中）。您可以将其视为调整特征映射 `k` 的整体亮度的旋钮。
- $w_{u,v,k',k}$ 是层 `L` 的特征映射 `k` 中的任何神经元与位于行 `u`，列 `v`（相对于神经元的感受野）的输入之间的连接权重，以及特征映射k'。

TensorFlow 实现

在 Tensorflow 中，每个输入图像的通常被表示为三维张量 `[height, width, channels]`。一个小批次被表示为四维张量 `[mini-batch size, height, width, channels]`。卷积层的权重被表示为四维张量 $[f_h, f_w, f_n, f_{n'}]$。卷积层的偏差项简单地表示为一维形

状的张量 $[f_n]$。我们来看一个简单的例子。 下面的代码使用 **Scikit-Learn** 的 `load_sample_images()`（加载两个彩色图像，一个中国庙宇，另一个是一朵花）加载两个样本图像。 然后创建两个 `7×7` 的卷积核（一个中间是垂直的白线，另一个是水平的白线），并将他们应用到两张图形中,使用 TensorFlow 的 `conv2d()` 函数构建的卷积图层（使用零填充且步幅为 2）。 最后，绘制其中一个结果特征映射（类似于图 13-5 中的右上图）。

```
from sklearn.datasets import load_sample_image
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf

if __name__ == '__main__':

    # Load sample images
    china = load_sample_image("china.jpg")
    flower = load_sample_image("flower.jpg")
    dataset = np.array([china, flower], dtype=np.float32)
    batch_size, height, width, channels = dataset.shape

    # Create 2 filters
    filters = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
    filters[:, 3, :, 0] = 1  # vertical line
    filters[3, :, :, 1] = 1  # horizontal line

    # Create a graph with input X plus a convolutional layer applying the 2 filters
    X = tf.placeholder(tf.float32, shape=(None, height, width, channels))
    convolution = tf.nn.conv2d(X, filters, strides=[1,2,2,1], padding="SAME")

    with tf.Session() as sess:
        output = sess.run(convolution, feed_dict={X: dataset})

    plt.imshow(output[0, :, :, 1], cmap="gray") # plot 1st image's 2nd feature map
    plt.show()
```

大部分代码是不言而喻的，但 `conv2d()` 这一行值得解释一下：

- `X` 是输入小批次（4D 张量，如前所述）
- 卷积核是应用的一组卷积核（也是一个 4D 张量，如前所述）。
- 步幅是一个四元素的一维数组，其中两个中间的值是垂直和水平的步幅（ `sh` 和 `sw` ）。第一个和最后一个元素现在必须等于 1。他们可能有一天会被用来指定批量步长（跳过一些实例）和频道步幅（跳过上一层的特征映射或通道）。
- `padding` 必须是 `"VALID"` 或 `"SAME"` ：
    - 如果设置为 `"VALID"` ，卷积层不使用零填充，并且可能会忽略输入图像底部和右侧的某些行和列，具体取决于步幅，如图 13-7 所示（为简单起见， 这里只显示水平尺寸，当然，垂直尺寸也适用相同的逻辑）
    - 如果设置为 `"SAME"` ，则卷积层在必要时使用零填充。 在这种情况下，输出神经元的数量等于输入神经元的数量除以该步幅，向上舍入（在这个例子中， `ceil(13/5)= 3` ）。 然后在输入周围尽可能均匀地添加零。

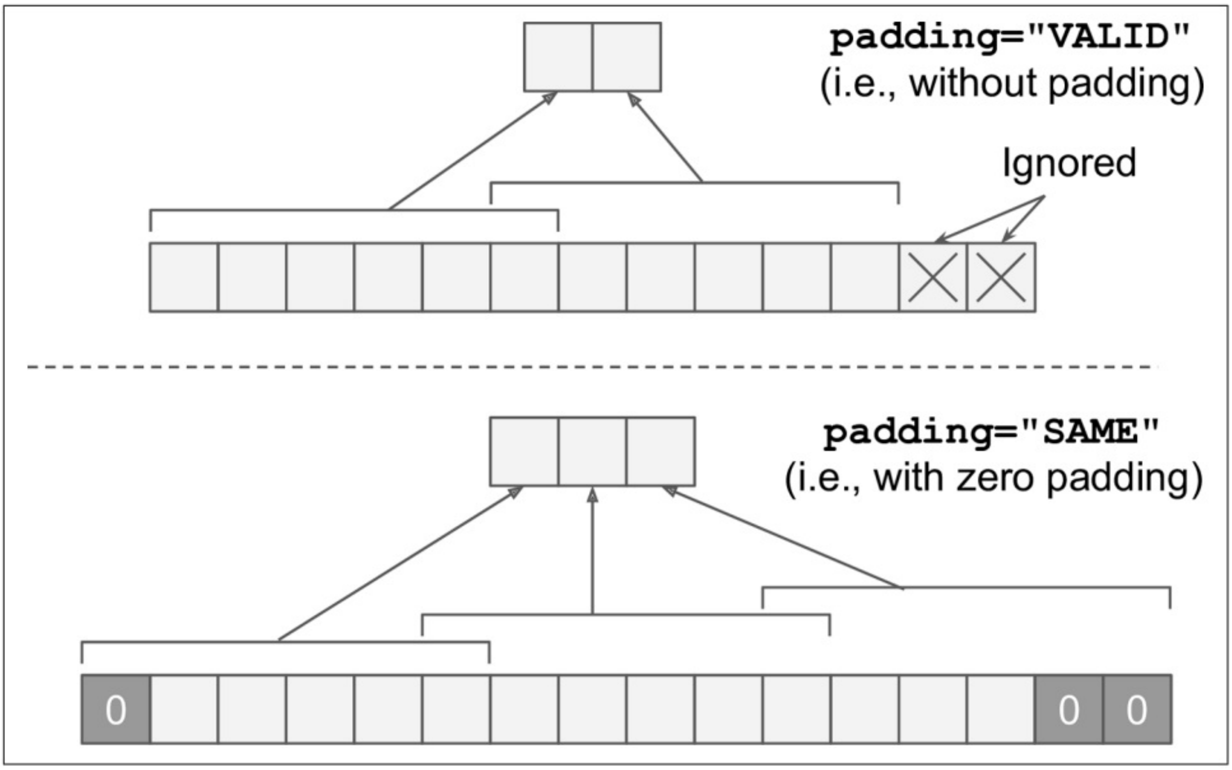


Figure 13-7. Padding options—input width: 13, filter width: 6, stride: 5

不幸的是，卷积图层有很多超参数：你必须选择卷积核的数量，高度和宽度，步幅和填充类型。 与往常一样，您可以使用交叉验证来查找正确的超参数值，但这非常耗时。 稍后我们将讨论常见的 CNN 体系结构，以便让您了解超参数值在实践中的最佳工作方式。

内存需求

CNN 的另一个问题是卷积层需要大量的 RAM，特别是在训练期间，因为反向传播需要在正向传递期间计算的所有中间值。

例如，考虑具有 `5×5` 卷积核的卷积层，输出 200 个尺寸为 `150×100` 的特征映射，步长为 1，使用 `SAME` 填充。 如果输入是 `150×100` RGB 图像（三个通道），则参数的数量是 `(5×5×3 + 1)×200 = 15,200` （ `+1` 对应于偏置项），这跟全连接层比较是相当小的。（具有 `150×100` 神经元的全连接层，每个连接到所有 `150×100×3` 输入，将具有 `150 ^ 2×100 ^ 2×3 = 675,000,000` 个参数！）然而，200 个特征映射中的每一个包含 `150×100` 个神经元，并且这些神经元中的每一个都需要计算其 `5×5×3 = 75` 个输入的权重和：总共 2.25 亿次浮点乘法。不像全连接层那么糟糕，但仍然是计算密集型的。 而且，如果使用 32 位浮点数来表示特征映射，则卷积层的输出将占用 RAM 的 `200×150×100×32 = 9600万` 位（大约 11.4MB）。 这只是一个例子！ 如果训练批次包含 100 个实例，则该层将占用超过 1 GB 的 RAM！

在推理过程中（即对新实例进行预测时），一旦下一层计算完毕，一层所占用的 RAM 就可以被释放，因此只需要两个连续层所需的 RAM 数量。 但是在训练期间，在正向传递期间计算的所有内容都需要被保留用于反向传递，所以所需的 RAM 量（至少）是所有层所需的 RAM

总量。

如果由于内存不足错误导致训练崩溃，则可以尝试减少小批量大小。 或者，您可以尝试使用步幅降低维度，或者删除几个图层。 或者你可以尝试使用 **16** 位浮点数而不是 **32** 位浮点数。 或者你可以在多个设备上分发 **CNN**。

池化层

一旦你理解了卷积层是如何工作的，池化层很容易掌握。 他们的目标是对输入图像进行二次抽样（即收缩）以减少计算负担，内存使用量和参数数量（从而限制过度拟合的风险）。 减少输入图像的大小也使得神经网络容忍一点点的图像变换（位置不变）。

就像在卷积图层中一样，池化层中的每个神经元都连接到前一层中有限数量的神经元的输出，位于一个小的矩形感受野内。 您必须像以前一样定义其大小，跨度和填充类型。 但是，汇集的神经元没有权重; 它所做的只是使用聚合函数（如最大值或平均值）来聚合输入。 图 13-8 显示了最大池层，这是最常见的池化类型。 在这个例子中，我们使用一个2×2的核，步幅为 2，没有填充。 请注意，只有每个核中的最大输入值才会进入下一层。 其他输入被丢弃。

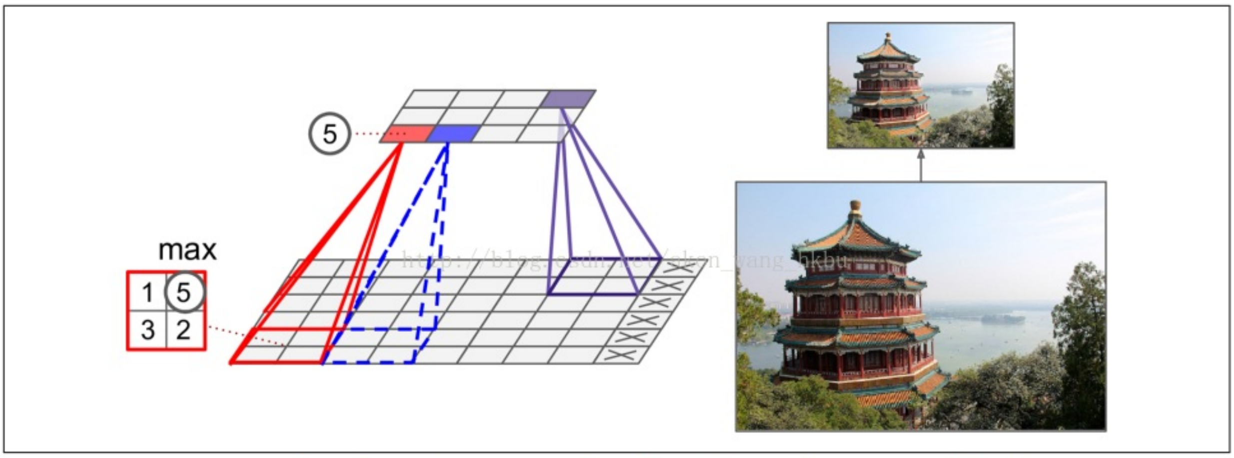


Figure 13-8. Max pooling layer (2 × 2 pooling kernel, stride 2, no padding)

这显然是一个非常具有破坏性的层：即使只有一个 `2×2` 的核和 2 的步幅，输出在两个方向上都会减小两倍（所以它的面积将减少四倍），一下减少了 75% 的输入值

池化层通常独立于每个输入通道工作，因此输出深度与输入深度相同。 接下来可以看到，在这种情况下，图像的空间维度（高度和宽度）保持不变，但是通道数目可以减少。

在 TensorFlow 中实现一个最大池层是非常容易的。 以下代码使用 `2×2` 核创建最大池化层，步幅为2，没有填充，然后将其应用于数据集中的所有图像：

```
import numpy as np
from sklearn.datasets import load_sample_image
import tensorflow as tf
import matplotlib.pyplot as plt

china = load_sample_image("china.jpg")
flower = load_sample_image("flower.jpg")

dataset = np.array([china, flower], dtype=np.float32)
batch_size, height, width, channels = dataset.shape

# Create 2 filters
filters = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
filters[:, 3, :, 0] = 1  # vertical line
filters[3, :, :, 1] = 1  # horizontal line

X = tf.placeholder(tf.float32, shape=(None, height, width, channels))
max_pool = tf.nn.max_pool(X, ksize=[1,2,2,1], strides=[1,2,2,1],padding="VALID")

with tf.Session() as sess:
    output = sess.run(max_pool, feed_dict={X: dataset})

plt.imshow(output[0].astype(np.uint8))  # plot the output for the 1st image
plt.show()
```

`ksize` 参数包含沿输入张量的所有四维的核形状： `[min-batch, height, width, channels]` 。TensorFlow 目前不支持在多个实例上合并，因此 `ksize` 的第一个元素必须等于 1。此外，它不支持在空间维度（高度和宽度）和深度维度上合并，因此 `ksize[1]` 和 `ksize[2]` 都必须等于 1，否则 `ksize[3]` 必须等于 1。

要创建一个平均池化层，只需使用 `avg_pool()` 函数而不是 `max_pool()` 。

现在你知道所有的构建模块来创建一个卷积神经网络。 我们来看看如何组装它们。

CNN 架构

典型的 CNN 体系结构有一些卷积层（每一个通常跟着一个 ReLU 层），然后是一个池化层，然后是另外几个卷积层（+ ReLU），然后是另一个池化层，等等。 随着网络的进展，图像变得越来越小，但是由于卷积层的缘故，图像通常也会越来越深（即更多的特征映射）（见图 13-9）。 在堆栈的顶部，添加由几个全连接层（+ ReLU）组成的常规前馈神经网络，并且最终层输出预测（例如，输出估计类别概率的 softmax 层）。

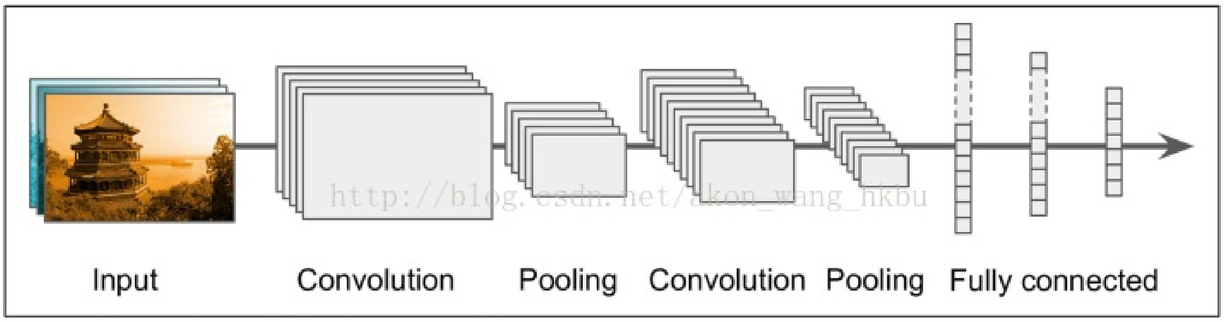


Figure 13-9. Typical CNN architecture

一个常见的错误是使用太大的卷积核。通常可以通过将两个 `3×3` 内核堆叠在一起来获得与 `9×9` 内核相同的效果，计算量更少。

多年来，这种基础架构的变体已经被开发出来，导致了该领域的惊人进步。这种进步的一个很好的衡量标准是比赛中的错误率，比如 ILSVRC ImageNet 的挑战。在这个比赛中，图像分类的五大误差率在五年内从 26% 下降到仅仅 3% 左右。前五位错误率是系统前5位预测未包含正确答案的测试图像的数量。图像很大（256 像素），有 1000 个类，其中一些非常微妙（尝试区分 120 个狗的品种）。查看获奖作品的演变是了解 CNN 如何工作的好方法。

我们先来看看经典的 LeNet-5 架构（1998 年），然后是 ILSVRC 挑战赛的三名获胜者 AlexNet（2012），GoogLeNet（2014）和 ResNet（2015）。

其他视觉任务在其他视觉任务中，如物体检测和定位以及图像分割，也取得了惊人的进展。在物体检测和定位中，神经网络通常输出图像中各种物体周围的一系列边界框。例如，参见 Maxine Oquab 等人的 2015 年论文，该论文为每个客体类别输出热图，或者 Russell Stewart 等人的 2015 年论文，该论文结合使用 CNN 来检测人脸，并使用递归神经网络来输出围绕它们的一系列边界框。在图像分割中，网络输出图像（通常与输入大小相同），其中每个像素指示相应输入像素所属的对象的类别。例如，查看 Evan Shelhamer 等人的 2016 年论文。

LeNet-5

LeNet-5 架构也许是最广为人知的 CNN 架构。如前所述，它是由 Yann LeCun 于 1998 年创建的，广泛用于手写数字识别（MNIST）。它由表 13-1 所示的层组成。

Table 13-1. LeNet-5 architecture

Layer	Type	Maps	Size	Kernel size	Stride	Activation
Out	Fully Connected	–	10	–	–	RBF
F6	Fully Connected	–	84	–	–	tanh
C5	Convolution	120	1 × 1	5 × 5	1	tanh
S4	Avg Pooling	16	5 × 5	2 × 2	2	tanh
C3	Convolution	16	10 × 10	5 × 5	1	tanh
S2	Avg Pooling	6	14 × 14	2 × 2	2	tanh
C1	Convolution	6	28 × 28	5 × 5	1	tanh
In	Input	1	32 × 32	–	–	–

有一些额外的细节要注意：

- MNIST 图像是 28×28 像素，但是它们被零填充到 32×32 像素，并且在被输入到网络之前被归一化。 网络的其余部分不使用任何填充，这就是为什么随着图像在网络中的进展，大小不断缩小。
- 平均池化层比平常稍微复杂一些：每个神经元计算输入的平均值，然后将结果乘以一个可学习的系数（每个特征映射一个），并添加一个可学习的偏差项（每个特征映射一个），然后最后应用激活函数。
- C3 图中的大多数神经元仅在三个或四个 S2 图（而不是全部六个 S2 图）中连接到神经元。有关详细信息，请参阅原始论文中的表 1。
- 输出层有点特殊：每个神经元不是计算输入和权向量的点积，而是输出其输入向量和其权向量之间的欧几里德距离的平方。 每个输出测量图像属于特定数字类别的多少。 交叉熵损失函数现在是首选，因为它更多地惩罚不好的预测，产生更大的梯度，从而更快地收敛。

Yann LeCun 的网站（“LENET”部分）展示了 LeNet-5 分类数字的很好的演示。

AlexNet

AlexNet CNN 架构赢得了 2012 年的 ImageNet ILSVRC 挑战赛：它达到了 17% 的 top-5 的错误率，而第二名错误率只有 26%！ 它由 Alex Krizhevsky（因此而得名），Ilya Sutskever 和 Geoffrey Hinton 开发。 它与 LeNet-5 非常相似，只是更大更深，它是第一个将卷积层直接堆叠在一起，而不是在每个卷积层顶部堆叠一个池化层。 表 13-2 介绍了这种架构。

Table 13-2. AlexNet architecture

Layer	Type	Maps	Size	Kernel size	Stride	Padding	Activation
Out	Fully Connected	–	1,000	–	–	–	Softmax
F9	Fully Connected	–	4,096	–	–	–	ReLU
F8	Fully Connected	–	4,096	–	–	–	ReLU
C7	Convolution	256	13 × 13	3 × 3	1	SAME	ReLU
C6	Convolution	384	13 × 13	3 × 3	1	SAME	ReLU
C5	Convolution	384	13 × 13	3 × 3	1	SAME	ReLU
S4	Max Pooling	256	13 × 13	3 × 3	2	VALID	–
C3	Convolution	256	27 × 27	5 × 5	1	SAME	ReLU
S2	Max Pooling	96	27 × 27	3 × 3	2	VALID	–
C1	Convolution	96	55 × 55	11 × 11	4	SAME	ReLU
In	Input	3 (RGB)	224 × 224	–	–	–	–

为了减少过拟合，作者使用了前面章节中讨论的两种正则化技术：首先他们在训练期间将丢失率（dropout 率为 50%）应用于层 F8 和 F9 的输出。其次，他们通过随机对训练图像进行各种偏移，水平翻转和改变照明条件来进行数据增强。

AlexNet 还在层 C1 和 C3 的 ReLU 步骤之后立即使用竞争标准化步骤，称为局部响应标准化（local response normalization）。 这种标准化形式使得在相同的位置的神经元被最强烈的激活但是在相邻的特征映射中抑制神经元（在生物神经元中观察到了这种竞争激活）。 这鼓励不同的特征映射特殊化，迫使它们分开，并让他们探索更广泛的特征，最终提升泛化能力。 公式 13-2 显示了如何应用 LRN。

Equation 13-2. Local response normalization

$$b_i = a_i \left(k + \alpha \sum_{j = j_{\text{low}}}^{j_{\text{high}}} {a_j}^2 \right)^{-\beta} \quad \text{with} \begin{cases} j_{\text{high}} = \min\left(i + \frac{r}{2}, f_n - 1\right) \\ j_{\text{low}} = \max\left(0, i - \frac{r}{2}\right) \end{cases}$$

- b_i 是位于特征映射 `i` 的神经元的标准化输出，在某行 `u` 和列 `v`（注意，在这个等式中我们只考虑位于这个行和列的神经元，所以 `u` 和 `v` 没有显示）。
- a_i 是在 ReLU 步骤之后，但在归一化之前的那个神经元的激活。
- `k`，`α`，`β` 和 `r` 是超参数。 `k` 称为偏置，`r` 称为深度半径。
- f_n 是特征映射的数量。

例如，如果 `r = 2` 且神经元具有强激活，则将抑制位于其上下的特征映射中的神经元的激活。

在 AlexNet 中，超参数设置如下：`r = 2`，`α= 0.00002`，`β= 0.75`，`k = 1`。这个步骤可以使用 TensorFlow 的 `local_response_normalization()` 操作来实现。

AlexNet 的一个名为 ZF Net 的变体由 Matthew Zeiler 和 Rob Fergus 开发，赢得了 2013 年 ILSVRC 的挑战。 它基本上是 AlexNet 的一些调整的超参数（特征映射的数量，内核大小，步幅等）。

GoogLeNet

GoogLeNet 架构是由 Christian Szegedy 等人开发的。 来自 Google Research，通过低于 7% 的 top-5 错误率，赢得了 ILSVRC 2014 的挑战赛。 这个伟大的表现很大程度上因为它比以前的 CNN 网络更深（见图 13-11）。 这是通过称为初始模块（inception modules）的子网

络实现的，这使得 GoogLeNet 比以前的架构更有效地使用参数：实际上，GoogLeNet 的参数比 AlexNet 少了 10 倍（约 600 万而不是 6000 万）。

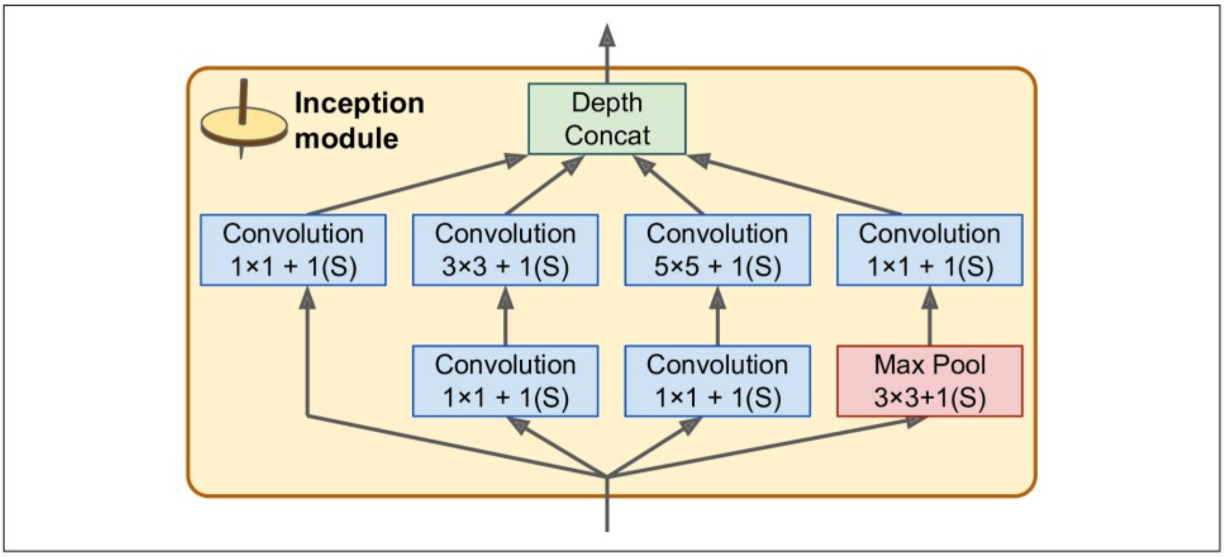


Figure 13-10. Inception module

初始模块的架构如图 13-10 所示。 符号 `3x3 + 2(S)` 表示该层使用 `3x3` 内核，步幅 2 和 `SAME` 填充。 输入信号首先被复制并馈送到四个不同的层。 所有卷积层都使用 ReLU 激活功能。 请注意，第二组卷积层使用不同的内核大小（ `1x1` ， `3x3` 和 `5x5` ），允许它们以不同的比例捕获图案。 还要注意，每一层都使用了跨度为1和SAME填充的（即使是最大的池化层），所以它们的输出全都具有与其输入相同的高度和宽度。这使得将所有输出在最后的深度连接层（depth concat layer）上沿着深度方向堆叠成为可能（即，堆叠来自所有四个顶部卷积层的特征映射）。 这个连接层可以在 TensorFlow 中使用 `concat()` 操作实现，其中 `axis = 3` （轴 3 是深度）。

您可能想知道为什么初始模块具有 `1x1` 内核的卷积层。 当然这些图层不能捕获任何功能，因为他们一次只能看一个像素？ 实际上，这些层次有两个目的：

首先，它们被配置为输出比输入少得多的特征映射，所以它们作为瓶颈层，意味着它们降低了维度。 在 `3x3` 和 `5x5` 卷积之前，这是特别有用的，因为这些在计算上是非常耗费内存的层。

其次，每一个卷积层对（ `[1 × 1, 3 × 3]` 和 `[1 × 1, 5 × 5]` 表现地像一个强大的卷积层，可以捕捉到更多的复杂的模式。事实上，这一对卷积层不是在图像上扫过一个简单的线性分类器（就像单个卷积层一样），而是在图像上扫描一个双层神经网络。

简而言之，您可以将整个初始模块视为类固醇卷积层，能够输出捕捉各种尺度复杂模式的特征映射。

每个卷积层的卷积核的数量是一个超参数。 不幸的是，这意味着你有六个超参数来调整你添加的每个初始层。

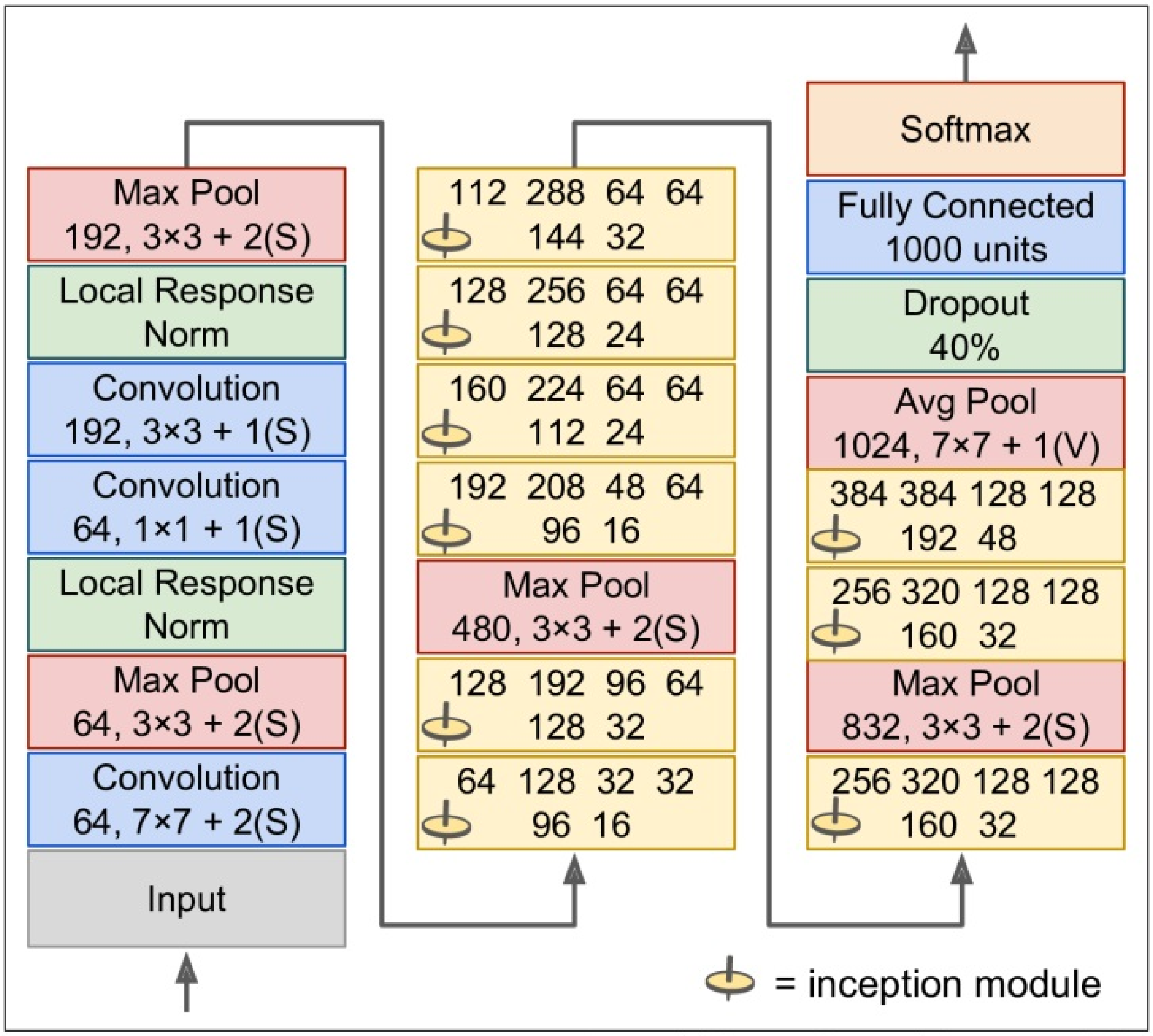


Figure 13-11. GoogLeNet architecture

现在

让我们来看看 GoogLeNet CNN 的架构（见图 13-11）。 它非常深，我们不得不将它分成三列，但是 GoogLeNet 实际上是一列，包括九个初始模块（带有旋转顶端的框），每个模块实际上包含三层。每个卷积层和池化层输出的特征映射的数量显示在内核大小前。 初始模块中的六个数字表示模块中每个卷积层输出的特征映射的数量（与图 13-10 中的顺序相同）。 请注意，所有的卷积层都使用 ReLU 激活函数。

让我们来过一遍这个网络：

- 前两层将图像的高度和宽度除以 4（使其面积除以 16），以减少计算负担。
- 然后，局部响应标准化层确保前面的层学习各种各样的功能（如前所述）
- 接下来是两个卷积层，其中第一个像瓶颈层一样。 正如前面所解释的，你可以把这一对看作是一个单一的更智能的卷积层。
- 再次，局部响应标准化层确保了先前的层捕捉各种各样的模式。
- 接下来，最大池化层将图像高度和宽度减少 2，再次加快计算速度。
- 然后是九个初始模块的堆叠，与几个最大池层交织，以降低维度并加速网络。

- 接下来，平均池化层使用具有 `VALID` 填充的特征映射的大小的内核，输出 `1×1` 特征映射：这种令人惊讶的策略被称为全局平均池化。 它有效地强制以前的图层产生特征映射，这些特征映射实际上是每个目标类的置信图（因为其他类型的功能将被平均步骤破坏）。 这样在 CNN 的顶部就不必有有几个全连接层（如 AlexNet），大大减少了网络中的参数数量，并减少了了过度拟合的风险。
- 最后一层是不言自明的：正则化 drop out，然后是具有 softmax 激活函数的完全连接层来输出估计类的概率。

这个图略有简化：原来的 GoogLeNet 架构还包括两个插在第三和第六个初始模块之上的辅助分类器。 它们都由一个平均池层，一个卷积层，两个全连接层和一个 softmax 激活层组成。 在训练期间，他们的损失（缩小了 70%）加在了整体损失上。 目标是解决消失梯度问题，正则化网络。 但是，结果显示其效果相对小。

ResNet

最后是，2015 年 ILSVRC 挑战赛的赢家 Kaiming He 等人开发的 Residual Network（或 ResNet），该网络的 top-5 误率低到惊人的 3.6%，它使用了一个非常深的 CNN，由 152 层组成。 能够训练如此深的网络的关键是使用跳过连接（skip connection，也称为快捷连接）：一个层的输入信号也被添加到位于下一层的输出。 让我们看看为什么这是有用的。

当训练一个神经网络时，目标是使其模拟一个目标函数 `h(x)` 。 如果将输入x添加到网络的输出中（即添加跳过连接），那么网络将被迫模拟 `f(x)= h(x) - x` 而不是 `h(x)` 。 这被称为残留学习（见图 13-12）。

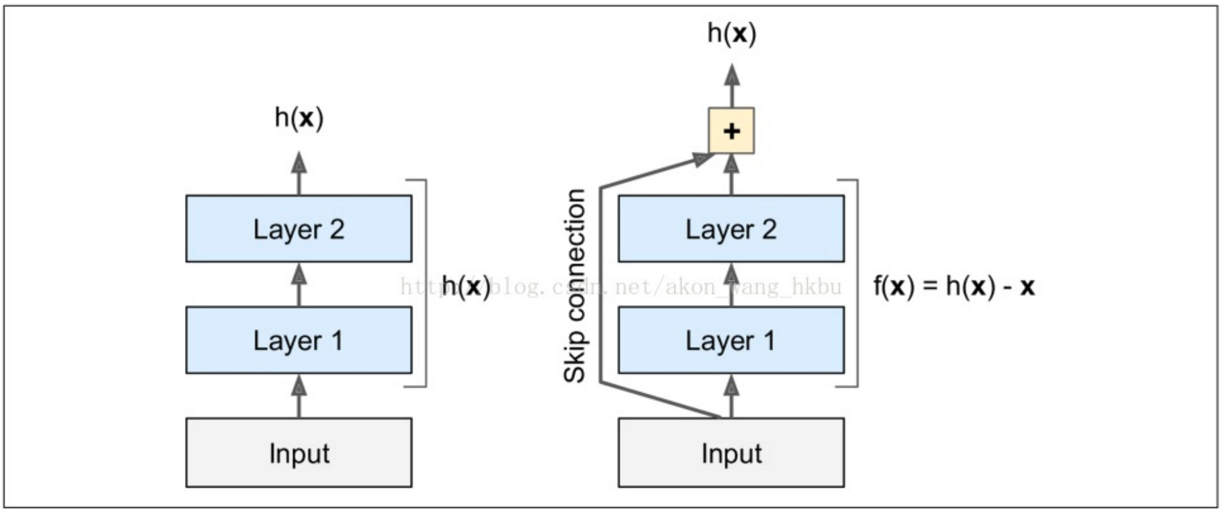


Figure 13-12. Residual learning

当你初始化一个普通的神经网络时，它的权重接近于零，所以网络只输出接近零的值。 如果添加跳过连接，则生成的网络只输出其输入的副本; 换句话说，它最初对身份函数进行建模。 如果目标函数与身份函数非常接近（常常是这种情况），这将大大加快训练速度。

而且，如果添加了许多跳转连接，即使几个层还没有开始学习，网络也可以开始进行（见图 13-13）。由于跳过连接，信号可以很容易地通过整个网络。深度剩余网络可以看作是一堆剩余单位，其中每个剩余单位是一个有跳过连接的小型神经网络。

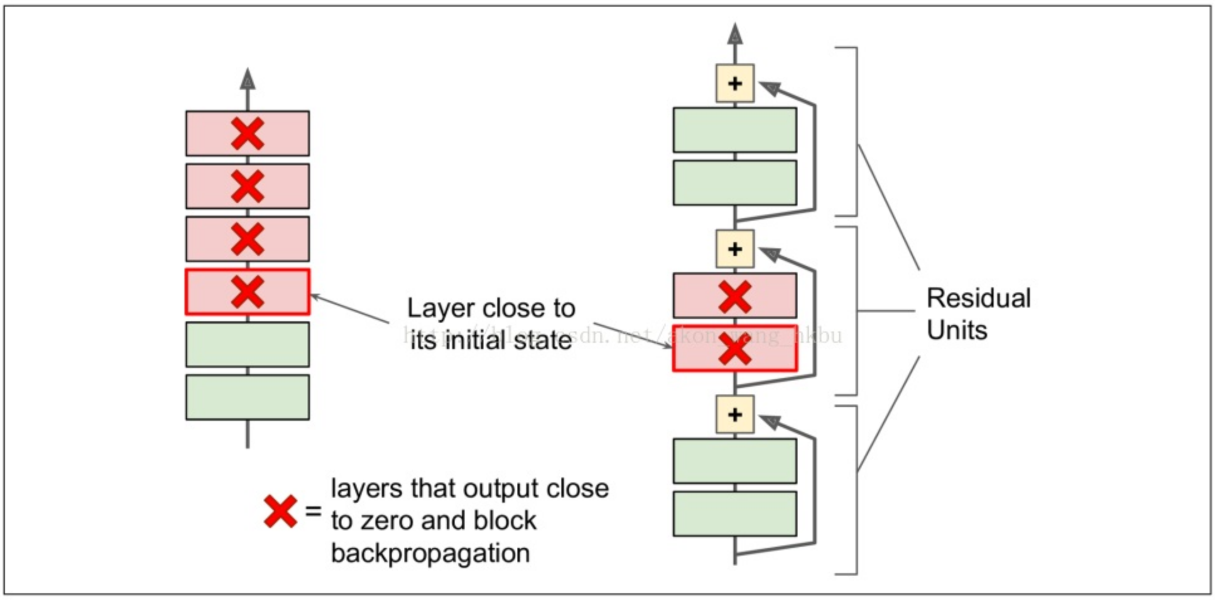


Figure 13-13. Regular deep neural network (left) and deep residual network (right)

现在让我们看看 ResNet 的架构（见图 13-14）。这实际上是令人惊讶的简单。它的开始和结束与GoogLeNet完全一样（除了没有 dropout 层），而在两者之间只是一堆很简单的残余单位。每个残差单元由两个卷积层组成，使用 `3x3` 的内核和保存空间维度（步幅 1，`SAME` 填充），批量归一化（BN）和 ReLU 激活。

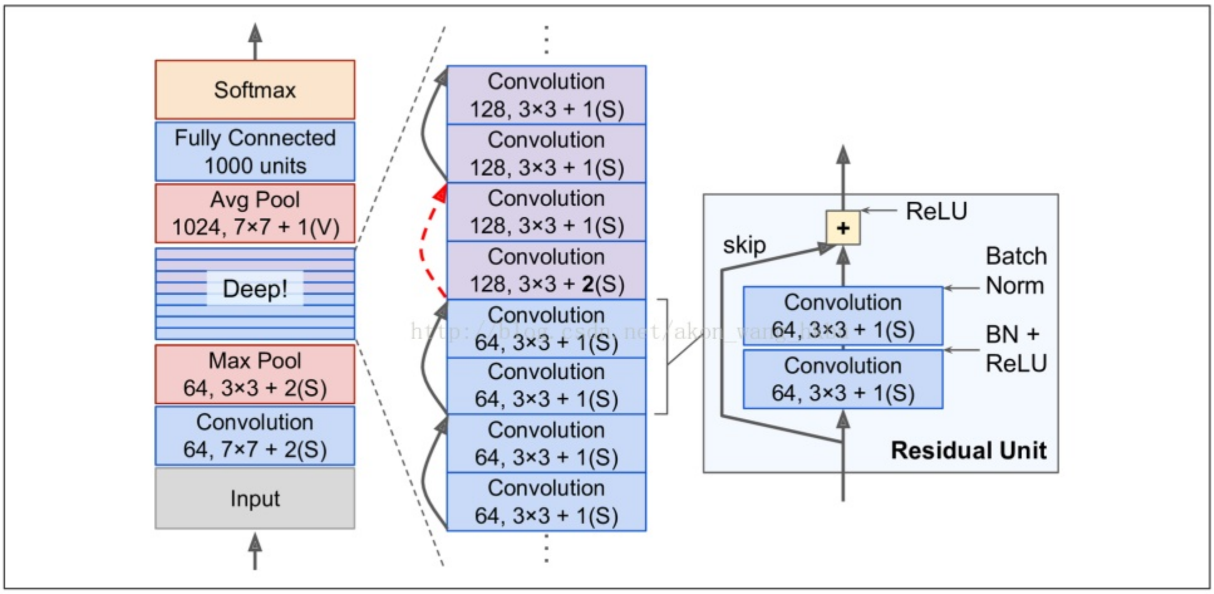


Figure 13-14. ResNet architecture

需要注意的是特征映射的数量每隔几个残差单位会加倍，同时它们的高度和宽度减半（使用步幅 2 卷积层）。发生这种情况时，输入不能直接添加到剩余单元的输出中，因为它们不具有相同的形状（例如，此问题影响图 13-14 中的虚线箭头表示的跳过连接）。为了解决这个问题，输入通过一个 `1x1` 卷积层，步长2和正确数量的输出特征映射（见图 13-15）。

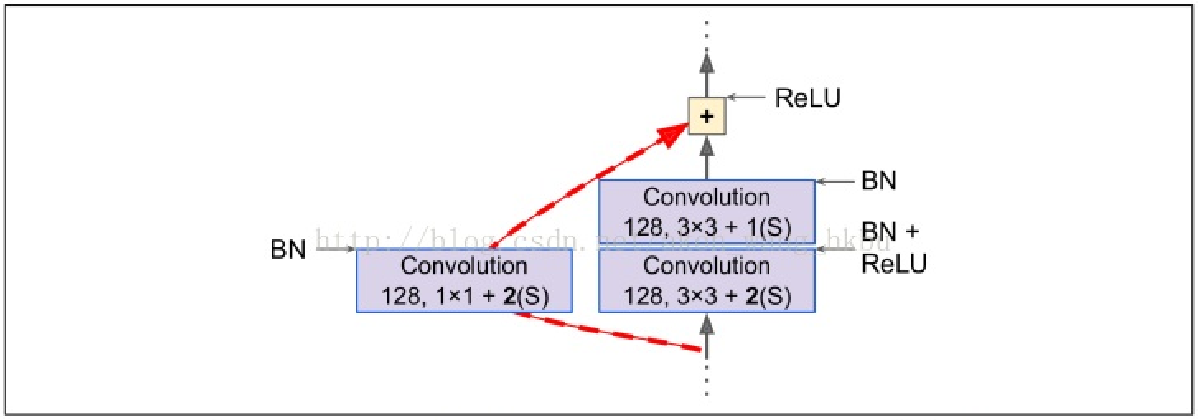


Figure 13-15. Skip connection when changing feature map size and depth

ResNet-34 是具有 34 个层（仅计算卷积层和完全连接层）的 ResNet，包含 3 个剩余单元输出 64 个特征映射，4 个剩余单元输出 128 个特征映射，6 个剩余单元输出 256 个特征映射，3 个剩余单元输出 512 个特征映射。

ResNet-152 更深，使用稍微不同的剩余单位。 他们使用三个卷积层，而不是两个 256 个特征映射的 `3×3` 的卷积层，它们使用三个卷积层：第一个卷积层只有 64 个特征映射（少 4 倍），这是一个瓶颈层（已经讨论过），然后是具有 64 个特征映射的 `3×3` 层，最后是具有 256 个特征映射（ `4×64` ）的另一个 `1×1` 卷积层，以恢复原始深度。ResNet-152 包含三个这样的剩余单位，输出 256 个特征映射，然后是 8 个剩余单位，输出 512 个特征映射，高达 36 个剩余单位，输出 1024 个特征映射，最后是 3 个剩余单位，输出 2048 个特征映射。

正如你所看到的，这个领域正在迅速发展，每年都会有各种各样的架构出现。 一个明显的趋势是 CNN 越来越深入。 他们也越来越轻量，需要越来越少的参数。 目前，ResNet 架构既是最强大的，也是最简单的，所以它现在应该是你应该使用的，但是每年都要继续关注 ILSVRC 的挑战。 2016 年获奖者是来自中国的 Trimps-Soushen 团队，他们的出错率惊人的缩减到 2.99%。 为了达到这个目标，他们训练了以前模型的组合，并将它们合并为一个整体。 根据任务的不同，降低的错误率可能会或可能不值得额外的复杂性。

还有其他一些架构可供您参考，特别是 VGGNet13（2014 年 ILSVRC 挑战赛的亚军）和 Inception-v414（将 GoGoLeNet 和 ResNet 的思想融合在一起，实现了接近 3% 的 top-5 误差 ImageNet 分类率）。

实施我们刚刚讨论的各种CNN架构真的没什么特别的。 我们之前看到如何构建所有的独立构建模块，所以现在您只需要组装它们来创建所需的构架。 我们将在即将开始的练习中构建 ResNet-34，您将在 Jupyter 笔记本中找到完整的工作代码。

TensorFlow 卷积操作

TensorFlow 还提供了一些其他类型的卷积层：

- `conv1d()` 为 1D 输入创建一个卷积层。 例如，在自然语言处理中这是有用的，其中句子可以表示为一维单词阵列，并且接受场覆盖一些邻近单词。

- `conv3d()` 创建一个 3D 输入的卷积层，如 3D PET 扫描。
- `atrous_conv2d()` 创建了一个 atrous 卷积层（“àtrous”是法语“with holes”）。 这相当于使用具有通过插入行和列（即，孔）而扩大的卷积核的普通卷积层。 例如，等于 `[[1,2,3]]` 的 `1×3` 卷积核可以以4的扩张率扩张，导致扩张的卷积核 `[[1,0,0,0,2,0,0,0,3]]` 。 这使得卷积层在没有计算价格的情况下具有更大的局部感受野，并且不使用额外的参数。
- `conv2d_transpose()` 创建了一个转置卷积层，有时称为去卷积层，它对图像进行上采样（这个名称是非常具有误导性的，因为这个层并不执行去卷积，这是一个定义良好的数学运算（卷积的逆））。这是通过在输入之间插入零来实现的，所以你可以把它看作是一个使用分数步长的普通卷积层。例如，在图像分割中，上采样是有用的：在典型的CNN中，特征映射越来越小当通过网络时，所以如果你想输出一个与输入大小相同的图像，你需要一个上采样层。
- `depthwise_conv2d()` 创建一个深度卷积层，将每个卷积核独立应用于每个单独的输入通道。 因此，如果有 `fn` 卷积核和 `fn'` 输入通道，那么这将输出 `fn×fn'` 特征映射。
- `separable_conv2d()` 创建一个可分离的卷积层，首先像深度卷积层一样工作，然后将 `1×1` 卷积层应用于结果特征映射。 这使得可以将卷积核应用于任意的输入通道组。

第14章 循环神经网络

击球手击出垒球，你会开始预测球的轨迹并立即开始奔跑。你追踪着它，不断调整你的移动步伐，最终在观众的一片雷鸣声中抓到它。无论是在听完朋友的话语还是早餐时预测咖啡的味道，你时刻在做的事就是在预测未来。在本章中，我们将讨论循环神经网络 -- 一类预测未来的网络（当然，是到目前为止）。它们可以分析时间序列数据，诸如股票价格，并告诉你什么时候买入和卖出。在自动驾驶系统中，他们可以预测行车轨迹，避免发生交通意外。更一般地说，它们可在任意长度的序列上工作，而不是截止目前我们讨论的只能在固定长度的输入上工作的网络。举个例子，它们可以把语句，文件，以及语音范本作为输入，使得它们在诸如自动翻译，语音到文本或者情感分析（例如，读取电影评论并提取评论者关于该电影的感觉）的自然语言处理系统中极为有用。

更近一步，循环神经网络的预测能力使得它们具备令人惊讶的创造力。你同样可以要求它们去预测一段旋律的下几个音符，然后随机选取这些音符的其中之一并演奏它。然后要求网络给出接下来最可能的音符，演奏它，如此周而复始。在你知道它之前，你的神经网络将创作一首诸如由谷歌 Magenta 工程所创造的《The one》的歌曲。类似的，循环神经网络可以生成语句，图像标注以及更多。目前结果还不能准确得到莎士比亚或者莫扎特的作品，但谁知道几年后他们能生成什么呢？

在本章中，我们将看到循环神经网络背后的基本概念，他们所面临的主要问题（换句话说，在第11章中讨论的消失／爆炸的梯度），以及广泛用于反抗这些问题的方法：LSTM 和 GRU cell（单元）。如同以往，沿着这个方式，我们将展示如何用 TensorFlow 实现循环神经网络。最终我们将看看及其翻译系统的架构。

循环神经元

到目前为止，我们主要关注的是前馈神经网络，其中激活仅从输入层到输出层的一个方向流动（附录 E 中的几个网络除外）。 循环神经网络看起来非常像一个前馈神经网络，除了它也有连接指向后方。 让我们看一下最简单的 RNN，它由一个神经元接收输入，产生一个输出，并将输出发送回自己，如图 14-1（左）所示。 在每个时间步 `t` （也称为一个帧），这个循环神经元接收输入 $x^{(t)}$ 以及它自己的前一时间步长 $y^{(t-1)}$ 的输出。 我们可以用时间轴来表示这个微小的网络，如图 14-1（右）所示。 这被称为随着时间的推移展开网络。

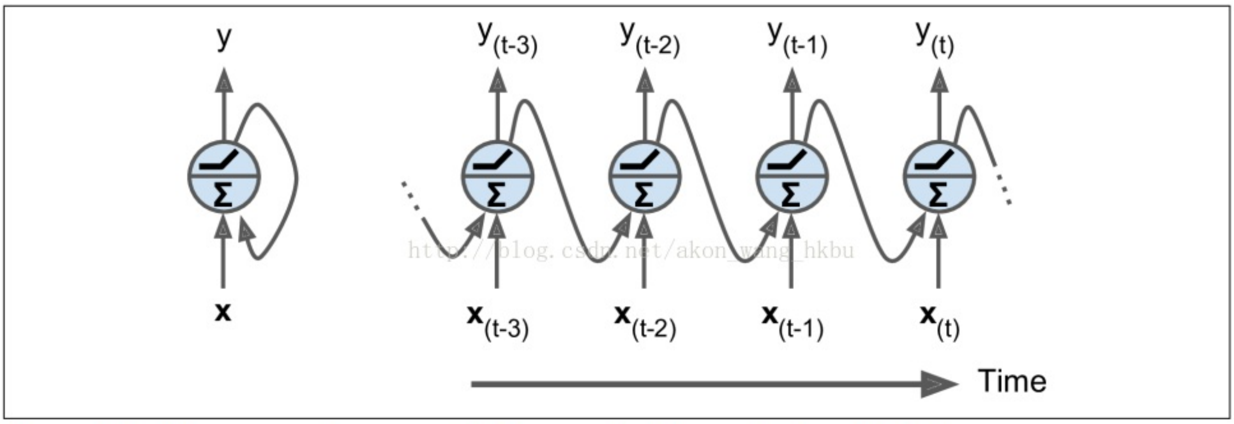


Figure 14-1. A recurrent neuron (left), unrolled through time (right)

你可以轻松创建一个循环神经元层。 在每个时间步t，每个神经元都接收输入向量 $x^{(t)}$ 和前一个时间步 $y^{(t-1)}$ 的输出向量，如图 14-2 所示。 请注意，输入和输出都是向量（当只有一个神经元时，输出是一个标量）。

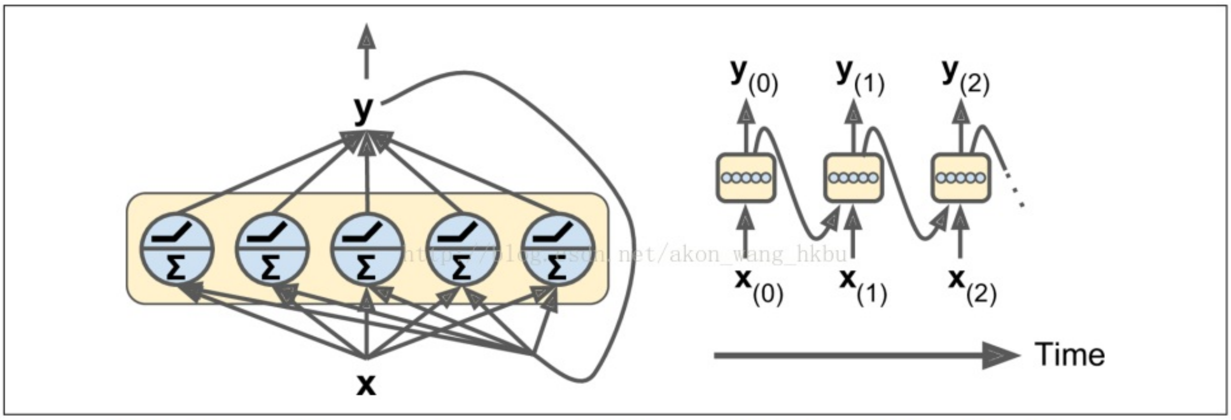


Figure 14-2. A layer of recurrent neurons (left), unrolled through time (right)

每个循环神经元有两组权重：一组用于输入 $x^{(t)}$，另一组用于前一时间步长 $y^{(t-1)}$ 的输出。我们称这些权重向量为 w_x 和 w_y。如公式 14-1 所示（ `b` 是偏差项， `φ(·)` 是激活函数，例如 ReLU），可以计算单个循环神经元的输出。

Equation 14-1. Output of a single recurrent neuron for a single instance

$$\mathbf{y}_{(t)} = \phi\left({\mathbf{x}_{(t)}}^T \cdot \mathbf{w}_x + {\mathbf{y}_{(t-1)}}^T \cdot \mathbf{w}_y + b\right)$$

就像前馈神经网络一样，我们可以使用上一个公式的向量化形式，对整个小批量计算整个层的输出（见公式 14-2）。

Equation 14-2. Outputs of a layer of recurrent neurons for all instances in a mini-batch

$$\mathbf{Y}_{(t)} = \phi\left(\mathbf{X}_{(t)} \cdot \mathbf{W}_x + \mathbf{Y}_{(t-1)} \cdot \mathbf{W}_y + \mathbf{b}\right)$$
$$= \phi\left(\left[\mathbf{X}_{(t)} \quad \mathbf{Y}_{(t-1)}\right] \cdot \mathbf{W} + \mathbf{b}\right) \text{ with } \mathbf{W} = \begin{bmatrix}\mathbf{W}_x \\ \mathbf{W}_y\end{bmatrix}$$

- $Y^{(t)}$ 是 $m \times n_{neurons}$ 矩阵，包含在最小批次中每个实例在时间步 `t` 处的层输出（`m` 是小批次中的实例数，$n_{neurons}$ 是神经元数）。
- $X^{(t)}$ 是 $m \times n_{inputs}$ 矩阵，包含所有实例的输入的 （n_{inputs} 是输入特征的数量）。
- W_x 是 $n_{inputs} \times n_{neurons}$ 矩阵，包含当前时间步的输入的连接权重的。
- W_y 是 $n_{neurons} \times n_{neurons}$ 矩阵，包含上一个时间步的输出的连接权重。
- 权重矩阵 W_x 和 W_y 通常连接成单个权重矩阵 `W`，形状为 $(n_{inputs} + n_{neurons}) \times n_{neurons}$（见公式 14-2 的第二行）
- `b` 是大小为 $n_{neurons}$ 的向量，包含每个神经元的偏置项。

注意，$Y^{(t)}$ 是 $X^{(t)}$ 和 $Y^{(t-1)}$ 的函数，它是 $X^{(t-1)}$ 和 $Y^{(t-2)}$ 的函数，它是 $X^{(t-2)}$ 和 $Y^{(t-3)}$ 的函数，等等。 这使得 $Y^{(t)}$ 是从时间 `t = 0` 开始的所有输入（即 $X^{(0)}$，$X^{(1)}$，...，$X^{(t)}$）的函数。 在第一个时间步，`t = 0`，没有以前的输出，所以它们通常被假定为全零。

记忆单元

由于时间 `t` 的循环神经元的输出，是由所有先前时间步骤计算出来的的函数，你可以说它有一种记忆形式。一个神经网络的一部分，跨越时间步长保留一些状态，称为存储单元（或简称为单元）。单个循环神经元或循环神经元层是非常基本的单元，但本章后面我们将介绍一些更为复杂和强大的单元类型。

一般情况下，时间步 `t` 处的单元状态，记为 $h^{(t)}$（`h` 代表“隐藏”），是该时间步的某些输入和前一时间步的状态的函数：$h^{(t)} = f(h^{(t-1)}, x^{(t)})$。 其在时间步 `t` 处的输出，表示为 $y^{(t)}$，也和前一状态和当前输入的函数有关。 在我们已经讨论过的基本单元的情况下，输出等于单元状态，但是在更复杂的单元中并不总是如此，如图 14-3 所示。

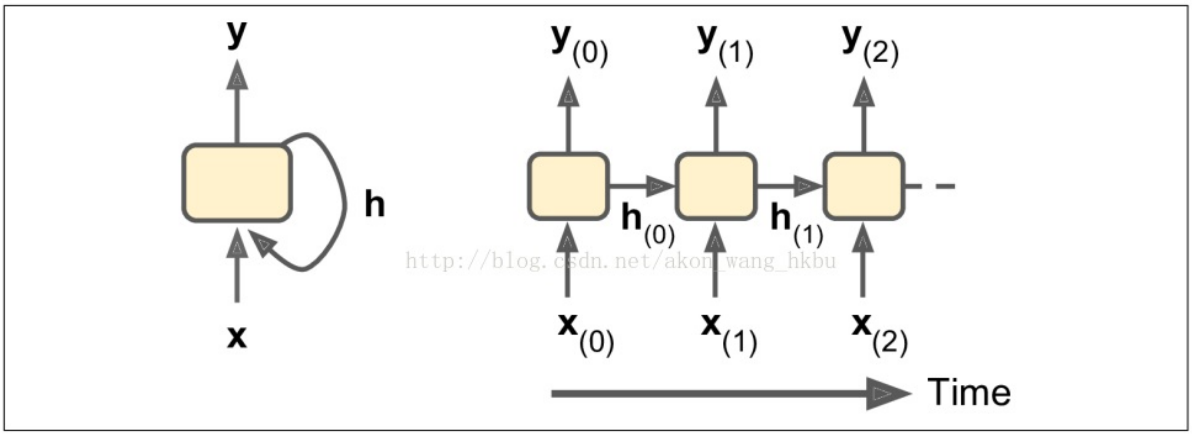


Figure 14-3. A cell's hidden state and its output may be different

输入和输出序列

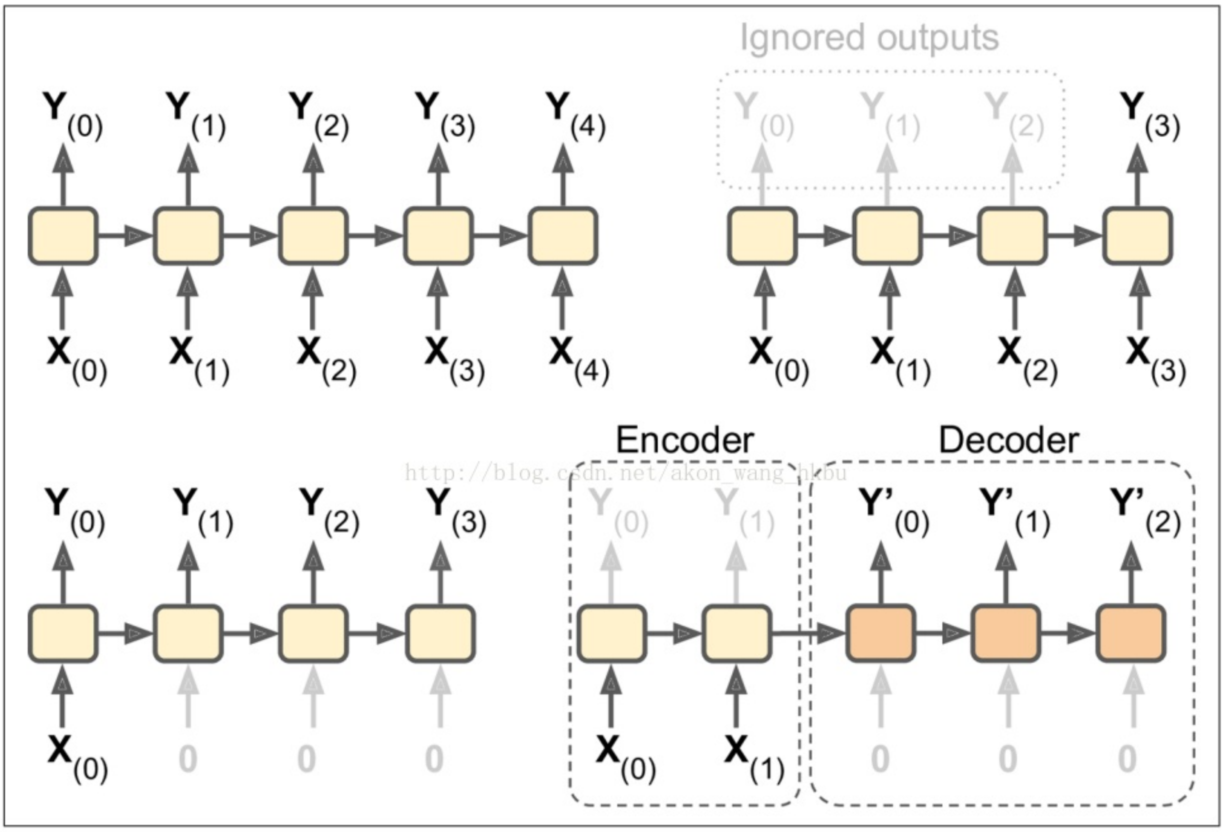


Figure 14-4. Seq to seq (top left), seq to vector (top right), vector to seq (bottom left), delayed seq to seq (bottom right)

RNN 可以同时进行一系列输入并产生一系列输出（见图 14-4，左上角的网络）。例如，这种类型的网络对于预测时间序列（如股票价格）非常有用：你在过去的 `N` 天内给出价格，并且它必须输出向未来一天移动的价格（即从 `N - 1` 天前到明天）。

或者，你可以向网络输入一系列输入，并忽略除最后一个之外的所有输出（请参阅右上角的网络）。 换句话说，这是一个向量网络的序列。 例如，你可以向网络提供与电影评论相对应的单词序列，并且网络将输出情感评分（例如，从 `-1 [恨]` 到 `+1 [爱]` ）。

相反，你可以在第一个时间步中为网络提供一个输入（而在其他所有时间步中为零），然后让它输出一个序列（请参阅左下角的网络）。 这是一个向量到序列的网络。 例如，输入可以是图像，输出可以是该图像的标题。

最后，你可以有一个序列到向量网络，称为编码器，后面跟着一个称为解码器的向量到序列网络（参见右下角的网络）。 例如，这可以用于将句子从一种语言翻译成另一种语言。 你会用一种语言给网络喂一个句子，编码器会把这个句子转换成单一的向量表示，然后解码器将这个向量解码成另一种语言的句子。 这种称为编码器 - 解码器的两步模型，比用单个序列到序列的 RNN（如左上方所示的那个）快速地进行翻译要好得多，因为句子的最后一个单词可以 影响翻译的第一句话，所以你需要等到听完整个句子才能翻译。

TensorFlow 中的基本 RNN

首先，我们来实现一个非常简单的 RNN 模型，而不使用任何 TensorFlow 的 RNN 操作，以更好地理解发生了什么。 我们将使用 tanh 激活函数创建由 5 个循环神经元的循环层组成的 RNN（如图 14-2 所示的 RNN）。 我们将假设 RNN 只运行两个时间步，每个时间步输入大小为 3 的向量。 下面的代码构建了这个 RNN，展开了两个时间步骤：

```
n_inputs = 3
n_neurons = 5
X0 = tf.placeholder(tf.float32, [None, n_inputs])
X1 = tf.placeholder(tf.float32, [None, n_inputs])
Wx = tf.Variable(tf.random_normal(shape=[n_inputs, n_neurons], dtype=tf.float32))
Wy = tf.Variable(tf.random_normal(shape=[n_neurons, n_neurons], dtype=tf.float32))
b = tf.Variable(tf.zeros([1, n_neurons], dtype=tf.float32))
Y0 = tf.tanh(tf.matmul(X0, Wx) + b)
Y1 = tf.tanh(tf.matmul(Y0, Wy) + tf.matmul(X1, Wx) + b)
init = tf.global_variables_initializer()
```

这个网络看起来很像一个双层前馈神经网络，有一些改动：首先，两个层共享相同的权重和偏差项，其次，我们在每一层都有输入，并从每个层获得输出。 为了运行模型，我们需要在两个时间步中都有输入，如下所示：

```
    # Mini-batch: instance 0,instance 1,instance 2,instance 3
    X0_batch = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 0, 1]])  # t = 0
    X1_batch = np.array([[9, 8, 7], [0, 0, 0], [6, 5, 4], [3, 2, 1]])  # t = 1
    with tf.Session() as sess:
        init.run()
    Y0_val, Y1_val = sess.run([Y0, Y1], feed_dict={X0: X0_batch, X1: X1_batch})
```

这个小批量包含四个实例，每个实例都有一个由两个输入组成的输入序列。 最后， `Y0_val` 和 `Y1_val` 在所有神经元和小批量中的所有实例的两个时间步中包含网络的输出：

```
>>> print(Y0_val) # output at t = 0
[[-0.2964572 0.82874775 -0.34216955 -0.75720584 0.19011548] # instance 0
[-0.12842922 0.99981797 0.84704727 -0.99570125 0.38665548] # instance 1
[ 0.04731077 0.99999976 0.99330056 -0.999933 0.55339795] # instance 2
[ 0.70323634 0.99309105 0.99909431 -0.85363263 0.7472108 ]] # instance 3
>>> print(Y1_val) # output at t = 1
[[ 0.51955646 1\. 0.99999022 -0.99984968 -0.24616946] # instance 0
[-0.70553327 -0.11918639 0.48885304 0.08917919 -0.26579669] # instance 1
[-0.32477224 0.99996376 0.99933046 -0.99711186 0.10981458] # instance 2
[-0.43738723 0.91517633 0.97817528 -0.91763324 0.11047263]] # instance 3
```

这并不难，但是当然如果你想能够运行 100 多个时间步骤的 RNN，这个图形将会非常大。现在让我们看看如何使用 TensorFlow 的 RNN 操作创建相同的模型。

完整代码

```
import numpy as np
import tensorflow as tf

if __name__ == '__main__':
    n_inputs = 3
    n_neurons = 5
    X0 = tf.placeholder(tf.float32, [None, n_inputs])
    X1 = tf.placeholder(tf.float32, [None, n_inputs])
    Wx = tf.Variable(tf.random_normal(shape=[n_inputs, n_neurons], dtype=tf.float32))
    Wy = tf.Variable(tf.random_normal(shape=[n_neurons, n_neurons], dtype=tf.float32))
    b = tf.Variable(tf.zeros([1, n_neurons], dtype=tf.float32))
    Y0 = tf.tanh(tf.matmul(X0, Wx) + b)
    Y1 = tf.tanh(tf.matmul(Y0, Wy) + tf.matmul(X1, Wx) + b)
    init = tf.global_variables_initializer()

    # Mini-batch: instance 0,instance 1,instance 2,instance 3
    X0_batch = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 0, 1]])  # t = 0
    X1_batch = np.array([[9, 8, 7], [0, 0, 0], [6, 5, 4], [3, 2, 1]])  # t = 1
    with tf.Session() as sess:
        init.run()
        Y0_val, Y1_val = sess.run([Y0, Y1], feed_dict={X0: X0_batch, X1: X1_batch})

    print(Y0_val,'\n')
    print(Y1_val)
```

时间上的静态展开

`static_rnn()` 函数通过链接单元来创建一个展开的 RNN 网络。下面的代码创建了与上一个完全相同的模型：

```
X0 = tf.placeholder(tf.float32, [None, n_inputs])
X1 = tf.placeholder(tf.float32, [None, n_inputs])

basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
output_seqs, states = tf.contrib.rnn.static_rnn(basic_cell, [X0, X1],
                                                dtype=tf.float32)
Y0, Y1 = output_seqs
```

首先，我们像以前一样创建输入占位符。 然后，我们创建一个 `BasicRNNCell` ，你可以将其视为一个工厂，创建单元的副本以构建展开的 RNN（每个时间步一个）。 然后我们调用 `static_rnn()` ，向它提供单元工厂和输入张量，并告诉它输入的数据类型（用来创建初始状态矩阵，默认情况下是全零）。 `static_rnn()` 函数为每个输入调用单元工厂的 `__call__()` 函数，创建单元的两个副本（每个单元包含 5 个循环神经元的循环层），并具有共享的权重和偏置项，像前面一样。 `static_rnn()` 函数返回两个对象。 第一个是包含每个时间步的输出张量的 Python 列表。 第二个是包含网络最终状态的张量。 当你使用基本的单元时，最后的状态就等于最后的输出。

如果有 50 个时间步长，则不得不定义 50 个输入占位符和 50 个输出张量。而且，在执行时，你将不得不为 50 个占位符中的每个占位符输入数据并且还要操纵 50 个输出。我们来简化一下。下面的代码再次构建相同的 RNN，但是这次它需要一个形状为 `[None，n_steps，n_inputs]` 的单个输入占位符，其中第一个维度是最小批量大小。然后提取每个时间步的输入序列列表。 `X_seqs` 是形状为 `n_steps` 的 Python 列表，包含形状为 `[None，n_inputs]` 的张量，其中第一个维度同样是最小批量大小。为此，我们首先使用 `transpose()` 函数交换前两个维度，以便时间步骤现在是第一维度。然后，我们使 `unstack()` 函数沿第一维（即每个时间步的一个张量）提取张量的 Python 列表。接下来的两行和以前一样。最后，我们使用 `stack()` 函数将所有输出张量合并成一个张量，然后我们交换前两个维度得到最终输出张量，形状为 `[None, n_steps，n_neurons]` （第一个维度是小批量大小）。

```
X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
X_seqs = tf.unstack(tf.transpose(X, perm=[1, 0, 2]))

basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
output_seqs, states = tf.contrib.rnn.static_rnn(basic_cell, X_seqs,
                                                dtype=tf.float32)
outputs = tf.transpose(tf.stack(output_seqs), perm=[1, 0, 2])
```

现在我们可以通过给它提供一个包含所有小批量序列的张量来运行网络：

```
X_batch = np.array([
        # t = 0      t = 1
        [[0, 1, 2], [9, 8, 7]], # instance 1
        [[3, 4, 5], [0, 0, 0]], # instance 2
        [[6, 7, 8], [6, 5, 4]], # instance 3
        [[9, 0, 1], [3, 2, 1]], # instance 4
    ])

with tf.Session() as sess:
    init.run()
    outputs_val = outputs.eval(feed_dict={X: X_batch})
```

我们得到所有实例，所有时间步长和所有神经元的单一 `outputs_val` 张量：

```
>>> print(outputs_val)
[[[-0.2964572   0.82874775 -0.34216955 -0.75720584  0.19011548]
  [ 0.51955646  1.          0.99999022 -0.99984968 -0.24616946]]

 [[-0.12842922  0.99981797  0.84704727 -0.99570125  0.38665548]
  [-0.70553327 -0.11918639  0.48885304  0.08917919 -0.26579669]]

 [[ 0.04731077  0.99999976  0.99330056 -0.999933    0.55339795]
  [-0.32477224  0.99996376  0.99933046 -0.99711186  0.10981458]]

 [[ 0.70323634  0.99309105  0.99909431 -0.85363263  0.7472108 ]
  [-0.43738723  0.91517633  0.97817528 -0.91763324  0.11047263]]]
```


但是，这种方法仍然会建立一个每个时间步包含一个单元的图。 如果有 50 个时间步，这个图看起来会非常难看。 这有点像写一个程序而没有使用循环（例如，`Y0 = f(0,X0)` ；`Y1 = f(Y0,X1)` ；`Y2 = f(Y1,X2)` ；...；`Y50 = f(Y49,X50)` ）。 如果使用大图，在反向传播期间（特别是在 GPU 内存有限的情况下），你甚至可能会发生内存不足（OOM）错误，因为它必须在正向传递期间存储所有张量值，以便可以使用它们在反向传播期间计算梯度。

幸运的是，有一个更好的解决方案：`dynamic_rnn()` 函数。

时间上的动态展开

`dynamic_rnn()` 函数使用 `while_loop()` 操作，在单元上运行适当的次数，如果要在反向传播期间将 GPU内 存交换到 CPU 内存，可以设置 `swap_memory = True` ，以避免内存不足错误。 方便的是，它还可以在每个时间步（形状为 `[None, n_steps, n_inputs]` ）接受所有输入的单个张量，并且在每个时间步（形状 `[None, n_steps, n_neurons]` ）上输出所有输出的单个张量。 没有必要堆叠，拆散或转置。 以下代码使用 `dynamic_rnn()` 函数创建与之前相同的 RNN。 这太好了！

完整代码

```
import numpy as np
import tensorflow as tf
import pandas as pd

if __name__ == '__main__':
    n_steps = 2
    n_inputs = 3
    n_neurons = 5

    X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])

    basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
    outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)

    init = tf.global_variables_initializer()

    X_batch = np.array([
        [[0, 1, 2], [9, 8, 7]],  # instance 1
        [[3, 4, 5], [0, 0, 0]],  # instance 2
        [[6, 7, 8], [6, 5, 4]],  # instance 3
        [[9, 0, 1], [3, 2, 1]],  # instance 4
    ])

    with tf.Session() as sess:
        init.run()
        outputs_val = outputs.eval(feed_dict={X: X_batch})

    print(outputs_val)
```

在反向传播期间，`while_loop()` 操作会执行相应的步骤：在正向传递期间存储每次迭代的张量值，以便在反向传递期间使用它们来计算梯度。

处理变长输入序列

到目前为止，我们只使用固定大小的输入序列（全部正好两个步长）。 如果输入序列具有可变长度（例如，像句子）呢？ 在这种情况下，你应该在调用 `dynamic_rnn()` （或 `static_rnn()` ）函数时设置 `sequence_length` 参数；它必须是一维张量，表示每个实例的输入序列的长度。 例如：

```
n_steps = 2
n_inputs = 3
n_neurons = 5

reset_graph()

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
```

```
seq_length = tf.placeholder(tf.int32, [None])
outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32,
                                    sequence_length=seq_length)
```

例如，假设第二个输入序列只包含一个输入而不是两个输入。 为了适应输入张量 `X` ，必须填充零向量（因为输入张量的第二维是最长序列的大小，即 2）

```
X_batch = np.array([
        # step 0     step 1
        [[0, 1, 2], [9, 8, 7]], # instance 1
        [[3, 4, 5], [0, 0, 0]], # instance 2 (padded with zero vectors)
        [[6, 7, 8], [6, 5, 4]], # instance 3
        [[9, 0, 1], [3, 2, 1]], # instance 4
    ])
seq_length_batch = np.array([2, 1, 2, 2])
```

当然，你现在需要为两个占位符 `X` 和 `seq_length` 提供值：

```
with tf.Session() as sess:
    init.run()
    outputs_val, states_val = sess.run(
        [outputs, states], feed_dict={X: X_batch, seq_length: seq_length_batch})
```

现在，RNN 输出序列长度的每个时间步都会输出零向量（查看第二个时间步的第二个输出）：

```
>>> print(outputs_val)
[[[-0.2964572   0.82874775 -0.34216955 -0.75720584  0.19011548]
  [ 0.51955646  1.          0.99999022 -0.99984968 -0.24616946]]  # final state

 [[-0.12842922  0.99981797  0.84704727 -0.99570125  0.38665548]   # final state
  [ 0.          0.          0.          0.          0.        ]]  # zero vector

 [[ 0.04731077  0.99999976  0.99330056 -0.999933    0.55339795]
  [-0.32477224  0.99996376  0.99933046 -0.99711186  0.10981458]]  # final state

 [[ 0.70323634  0.99309105  0.99909431 -0.85363263  0.7472108 ]
  [-0.43738723  0.91517633  0.97817528 -0.91763324  0.11047263]]] # final state
```


此外，状态张量包含每个单元的最终状态（不包括零向量）：

```
>>> print(states_val)
[[ 0.51955646  1.          0.99999022 -0.99984968 -0.24616946]    # t = 1
 [-0.12842922  0.99981797  0.84704727 -0.99570125  0.38665548]    # t = 0 !!!
 [-0.32477224  0.99996376  0.99933046 -0.99711186  0.10981458]    # t = 1
 [-0.43738723  0.91517633  0.97817528 -0.91763324  0.11047263]]   # t = 1
```


处理变长输出序列

如果输出序列长度不一样呢？如果事先知道每个序列的长度（例如，如果知道长度与输入序列的长度相同），那么可以按照上面所述设置 `sequence_length` 参数。不幸的是，通常这是不可能的：例如，翻译后的句子的长度通常与输入句子的长度不同。在这种情况下，最常见的解决方案是定义一个称为序列结束标记（EOS 标记）的特殊输出。任何在 EOS 后面的输出应该被忽略（我们将在本章稍后讨论）。

好，现在你知道如何建立一个 RNN 网络（或者更准确地说是一个随着时间的推移而展开的 RNN 网络）。但是你怎么训练呢？

训练 RNN

为了训练一个 RNN，诀窍是在时间上展开（就像我们刚刚做的那样），然后简单地使用常规反向传播（见图 14-5）。这个策略被称为时间上的反向传播（BPTT）。

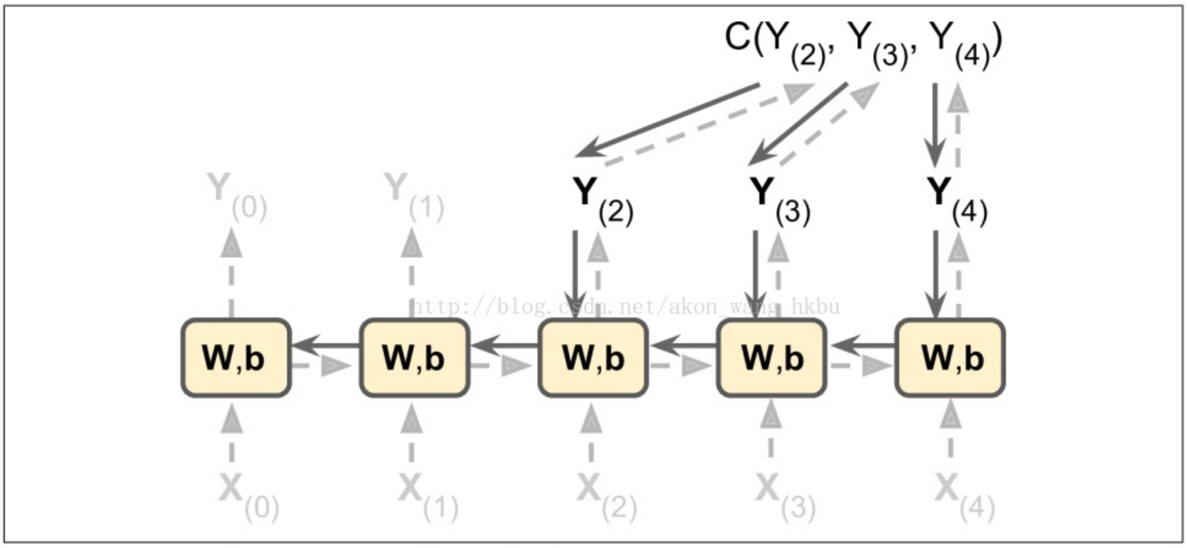


Figure 14-5. Backpropagation through time

就像在正常的反向传播中一样，展开的网络（用虚线箭头表示）有第一个正向传递。然后使用损失函数评估输出序列 $C\left(\mathbf{Y}_{(t_{\min})}, \mathbf{Y}_{(t_{\min}+1)}, \cdots, \mathbf{Y}_{(t_{\max})}\right)$ （其中 t_{min} 和 t_{max} 是第一个和最后一个输出时间步长，不计算忽略的输出），并且该损失函数的梯度通过展开的网络向后传播（实线箭头）；最后使用在 BPTT 期间计算的梯度来更新模型参数。请注意，梯度在损失函数所使用的所有输出中反向流动，而不仅仅通过最终输出（例如，在图 14-5 中，损失函数使用网络的最后三个输出 $Y^{(2)}$，$Y^{(3)}$ 和 $Y^{(4)}$，所以梯度流经这三个输出，但不通过 $Y^{(0)}$ 和 $Y^{(1)}$）。而且，由于在每个时间步骤使用相同的参数 `W` 和 `b`，所以反向传播将做正确的事情并且总结所有时间步骤。

训练序列分类器

我们训练一个 RNN 来分类 MNIST 图像。卷积神经网络将更适合于图像分类（见第 13 章），但这是一个你已经熟悉的简单例子。我们将把每个图像视为 28 行 28 像素的序列（因为每个MNIST图像是 `28×28` 像素）。我们将使用 150 个循环神经元的单元，再加上一个全连接层，其中包含连接到上一个时间步的输出的 10 个神经元（每个类一个），然后是一个 softmax 层（见图 14-6）。

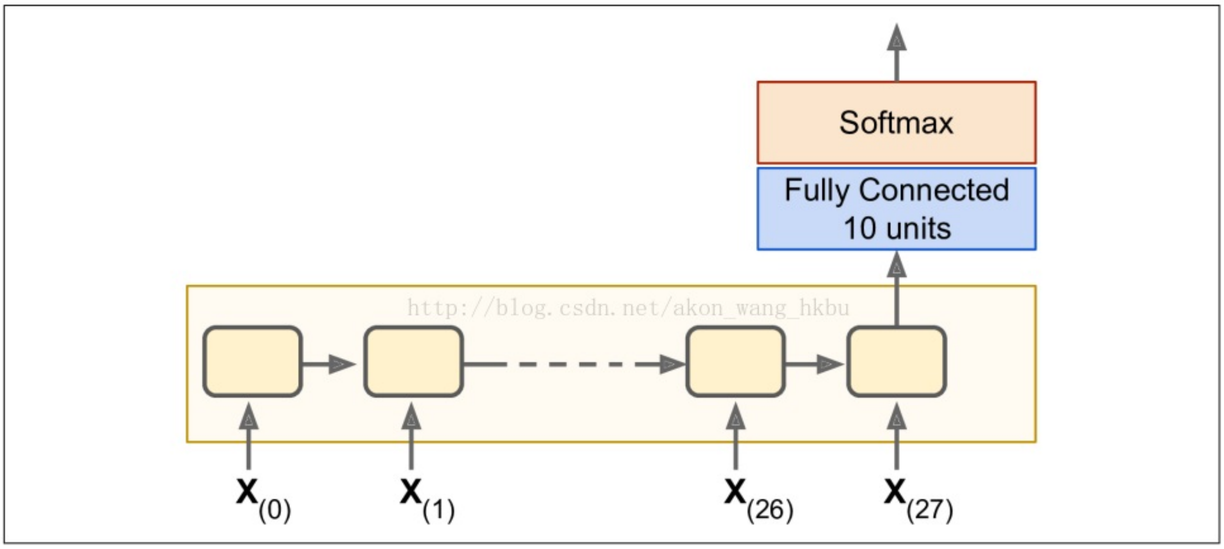


Figure 14-6. Sequence classifier

建模阶段非常简单，它和我们在第 10 章中建立的 MNIST 分类器几乎是一样的，只是展开的 RNN 替换了隐层。注意，全连接层连接到状态张量，其仅包含 RNN 的最终状态（即，第 28 个输出）。另请注意，`y` 是目标类的占位符。

```
n_steps = 28
n_inputs = 28
n_neurons = 150
n_outputs = 10

learning_rate = 0.001

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.int32, [None])

basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)

logits = tf.layers.dense(states, n_outputs)
xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y,
                                                          logits=logits)
loss = tf.reduce_mean(xentropy)
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(loss)
correct = tf.nn.in_top_k(logits, y, 1)
accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

init = tf.global_variables_initializer()
```

现在让我们加载 MNIST 数据，并按照网络的预期方式将测试数据重塑为 `[batch_size, n_steps, n_inputs]`。我们之后会关注训练数据的重塑。

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/")
X_test = mnist.test.images.reshape((-1, n_steps, n_inputs))
y_test = mnist.test.labels
```

现在我们准备训练 RNN 了。执行阶段与第 10 章中 MNIST 分类器的执行阶段完全相同，不同之处在于我们在将每个训练的批量提供给网络之前要重新调整。

```
batch_size = 150

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            X_batch = X_batch.reshape((-1, n_steps, n_inputs))
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
        acc_test = accuracy.eval(feed_dict={X: X_test, y: y_test})
        print(epoch, "Train accuracy:", acc_train, "Test accuracy:", acc_test)
```

输出应该是这样的：

```
0 Train accuracy: 0.713333 Test accuracy: 0.7299
1 Train accuracy: 0.766667 Test accuracy: 0.7977
...
98 Train accuracy: 0.986667 Test accuracy: 0.9777
99 Train accuracy: 0.986667 Test accuracy: 0.9809
```


我们获得了超过 98% 的准确性 - 不错！ 另外，通过调整超参数，使用 He 初始化初始化 RNN 权重，更长时间训练或添加一些正则化（例如，droupout），你肯定会获得更好的结果。

你可以通过将其构造代码包装在一个变量作用域内（例如，使用 `variable_scope("rnn", initializer = variance_scaling_initializer())` 来使用 He 初始化）来为 RNN 指定初始化器。

为预测时间序列而训练

现在让我们来看看如何处理时间序列，如股价，气温，脑电波模式等等。 在本节中，我们将训练一个 RNN 来预测生成的时间序列中的下一个值。 每个训练实例是从时间序列中随机选取的 20 个连续值的序列，目标序列与输入序列相同，除了向后移动一个时间步（参见图14-7）。

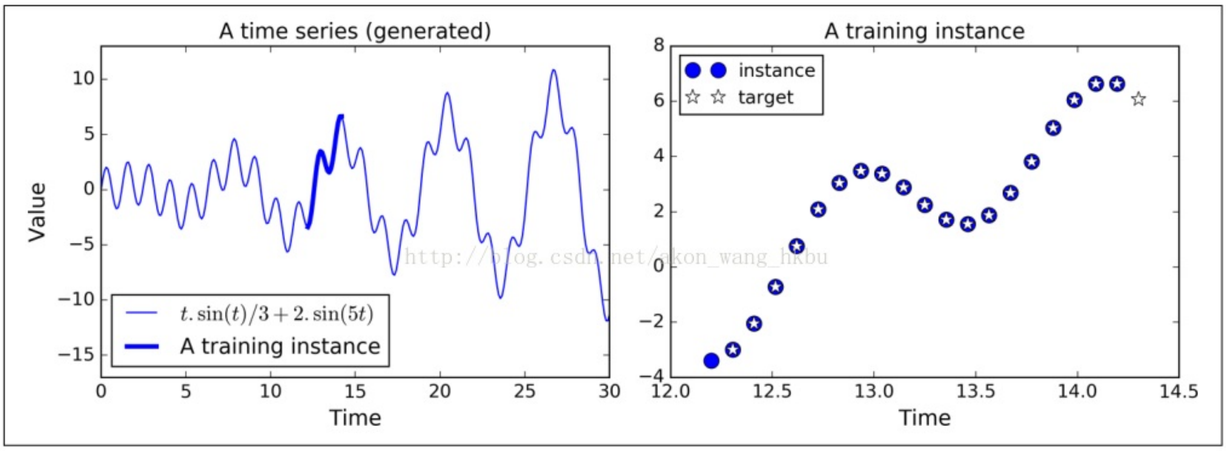


Figure 14-7. Time series (left), and a training instance from that series (right)

首先，我们来创建一个 RNN。 它将包含 100 个循环神经元，并且我们将在 20 个时间步骤上展开它，因为每个训练实例将是 20 个输入那么长。 每个输入将仅包含一个特征（在该时间的值）。 目标也是 20 个输入的序列，每个输入包含一个值。 代码与之前几乎相同：

```
n_steps = 20
n_inputs = 1
n_neurons = 100
n_outputs = 1

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.float32, [None, n_steps, n_outputs])
cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons, activation=tf.nn.relu)
outputs, states = tf.nn.dynamic_rnn(cell, X, dtype=tf.float32)
```

一般来说，你将不只有一个输入功能。 例如，如果你试图预测股票价格，则你可能在每个时间步骤都会有许多其他输入功能，例如竞争股票的价格，分析师的评级或可能帮助系统进行预测的任何其他功能。

在每个时间步，我们现在有一个大小为 100 的输出向量。但是我们实际需要的是每个时间步的单个输出值。 最简单的解决方法是将单元包装在 `OutputProjectionWrapper` 中。 单元包装器就像一个普通的单元，代理每个方法调用一个底层单元，但是它也增加了一些功能。 `Out putProjectionWrapper` 在每个输出之上添加一个完全连接的线性神经元层（即没有任何激活函数）（但不影响单元状态）。 所有这些完全连接的层共享相同（可训练）的权重和偏差项。 结果 RNN 如图 14-8 所示。

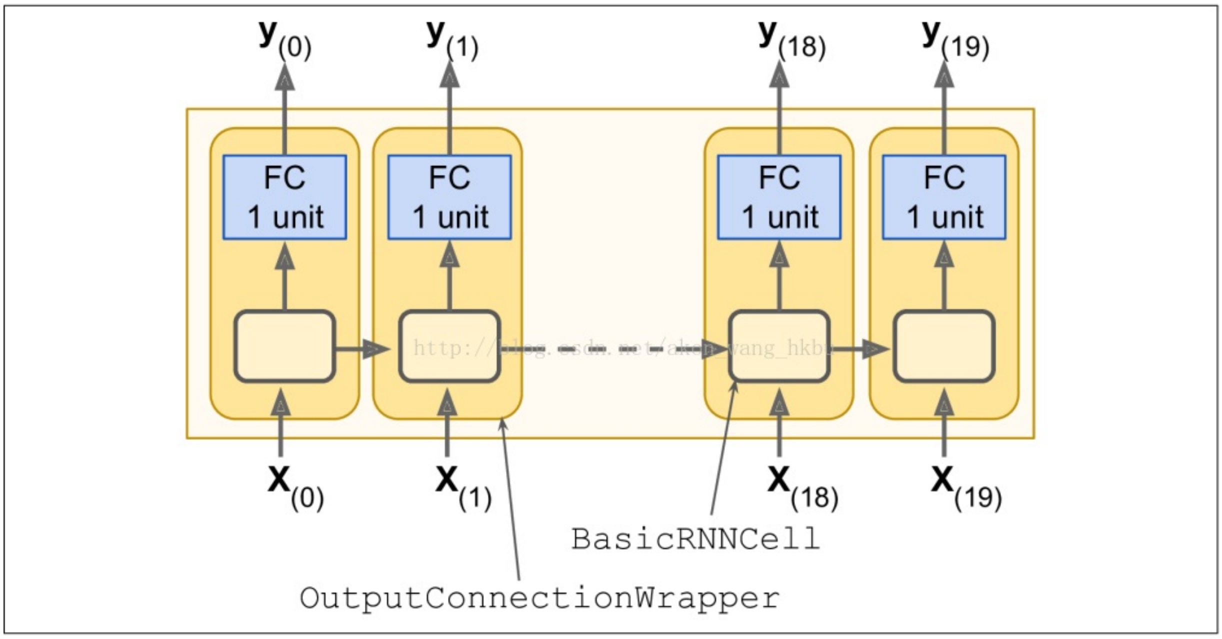


Figure 14-8. RNN cells using output projections

包装单元是相当容易的。让我们通过将 `BasicRNNCell` 包装到 `OutputProjectionWrapper` 中来调整前面的代码：

```
cell =tf.contrib.rnn.OutputProjectionWrapper(
    tf.contrib.rnn.BasicRNNCell(num_units=n_neurons,activation=tf.nn.relu),
    output_size=n_outputs)
```

到现在为止还挺好。现在我们需要定义损失函数。我们将使用均方误差（MSE），就像我们在之前的回归任务中所做的那样。接下来，我们将像往常一样创建一个 Adam 优化器，训练操作和变量初始化操作：

生成 RNN

到现在为止，我们已经训练了一个能够预测未来时刻样本值的模型，正如前文所述，可以用模型来生成新的序列。

为模型提供 长度为 `n_steps` 的种子序列, 比如全零序列，然后通过模型预测下一时刻的值；把该预测值添加到种子序列的末尾，用最后面 长度为 `n_steps` 的序列做为新的种子序列，做下一次预测，以此类推生成预测序列。

如图 14-11 所示，这个过程产生的序列会跟原始时间序列相似。

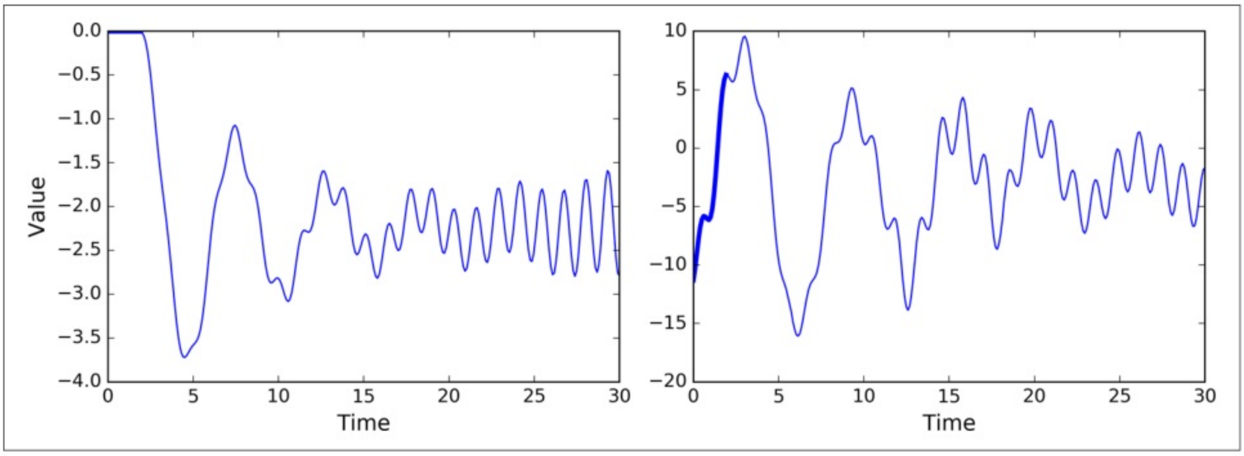


Figure 14-11. Creative sequences, seeded with zeros (left) or with an instance (right)

```
sequence = [0.] * n_steps
for iteration in range(300):
    X_batch = np.array(sequence[-n_steps:].reshape(1, n_steps, 1)
    y_pred = sess.run(outputs, feed_dict={X: X_batch}
    sequence.append(y_pred[0, -1, 0]
```

如果你试图把约翰·列侬的唱片塞给一个 RNN 模型，看它能不能生成下一张《想象》专辑。

> 注
>
> 约翰·列侬 有一张专辑《Imagine》（1971），这里取其双关的意思

也许你需要一个更强大的 RNN 网络，它有更多的神经元，层数也更多。下面来探究一下深度 RNN。

深度 RNN

一个朴素的想法就是把一层层神经元堆叠起来，正如图 14-12 所示的那样，它呈现了一种深度 RNN。

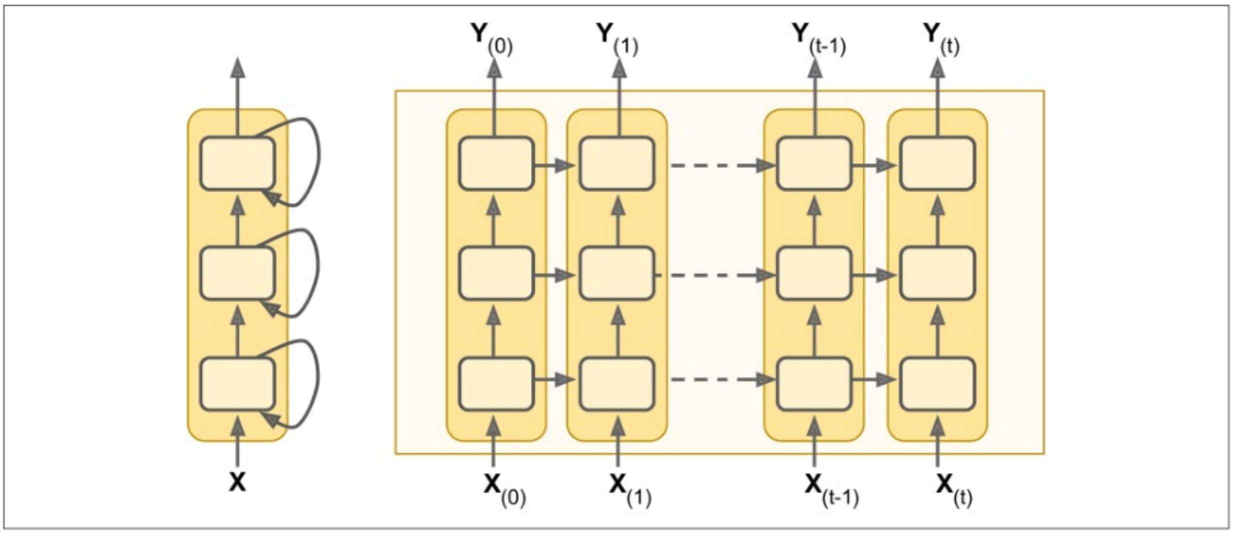


Figure 14-12. Deep RNN (left), unrolled through time (right)

为了用 TensorFlow 实现深度 RNN，可先创建一些神经单元，然后堆叠进 `MultiRNNCell` 。

以下代码中创建了 3 个相同的神经单元（当然也可以用不同类别的、包含不同不同数量神经元的单元）

```
n_neurons = 100
n_layers = 3

basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
multi_layer_cell = tf.contrib.rnn.MultiRNNCell([basic_cell] * n_layers)
outputs, states = tf.nn.dynamic_rnn(multi_layer_cell, X, dtype=tf.float32)
```

这些代码就完成了这部分堆叠工作。 `status` 变量包含了每层的一个张量，这个张量就代表了该层神经单元的最终状态（维度为 `[batch_size, n_neurons]` ）。

如果在创建 `MultiRNNCell` 时设置了 `state_is_tuple=False` ，那么 `status` 变量就变成了单个张量，它包含了每一层的状态，其在列的方向上进行了聚合，维度为 `[batch_size, n_layers*n_neurons]` 。

注意在 TensorFlow 版本 0.11.0 之前， `status` 是单个张量是默认设置。

在多个 GPU 上分布式部署深度 RNN 网络

Dropout 的应用

对于深层深度 RNN，在训练集上很容易过拟合。Dropout 是防止过拟合的常用技术。

可以简单的在 RNN 层之前或之后添加一层 Dropout 层，但如果需要在 RNN 层之间应用 Dropout 技术就需要 `DropoutWrapper` 。

下面的代码中，每一层的 RNN 的输入前都应用了 Dropout，Dropout 的概率为 50%。

```
keep_prob = 0.5

cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
cell_drop = tf.contrib.rnn.DropoutWrapper(cell, input_keep_prob=keep_prob)
multi_layer_cell = tf.contrib.rnn.MultiRNNCell([cell_drop]*n_layers)
rnn_outputs, states = tf.nn.dynamic_rnn(multi_layer_cell, X, dtype=tf.float32)
```

同时也可以通过设置 `output_keep_prob` 来在输出应用 Dropout 技术。

然而在以上代码中存在的主要问题是，Dropout 不管是在训练还是测试时都起作用了，而我们想要的仅仅是在训练时应用 Dropout。

很不幸的是 `DropoutWrapper` 不支持 `is_training` 这样一个设置选项。因此必须自己写 Dropout 包装类，或者创建两个计算图，一个用来训练，一个用来测试。后则可通过如下面代码这样实现。

```
import sys
is_training  = (sys.argv[-1] == "train")

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.float32, [None, n_steps, n_outputs])
cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
if is_training:
    cell = tf.contrib.rnn.DropoutWrapper(cell, input_keep_prob=keep_prob)
multi_layer_cell = tf.contrib.rnn.MultiRNNCell([cell]*n_layers)
rnn_outpus, status = tf.nn.dynamic_rnn(multi_layer_cell, X, dtype=tf.float32)

[...] # bulid the rest of the graph
init = tf.global_variables_initializer()
saver = tf.train.Saver()

with tf.Session() as sess:
    if is_training:
        init.run()
        for iteration in range(n_iterations):
            [...] # train the model
        save_path = saver.save(sess, "/tmp/my_model.ckpt")
    else:
        saver.restore(sess, "/tmp/my_model.ckpt")
        [...] # use the model
```

通过以上的方法就能够训练各种 RNN 网络了。然而对于长序列的 RNN 训练还言之过早，事情会变得有一些困难。

那么我们来探讨一下究竟这是为什么和怎么应对呢？

长时训练的困难

在训练长序列的 RNN 模型时，那么就需要把 RNN 在时间维度上展开成很深的神经网络。正如任何深度神经网络一样，其面临着梯度消失/爆炸的问题，使训练无法终止或收敛。

很多之前讨论过的缓解这种问题的技巧都可以应用在深度展开的 RNN 网络：好的参数初始化方式，非饱和的激活函数（如 ReLU），批量规范化（Batch Normalization），梯度截断（Gradient Clipping），更快的优化器。

即便如此，RNN 在处理适中的长序列（如 100 输入序列）也在训练时表现的很慢。

最简单和常见的方法解决训练时长问题就是在训练阶段仅仅展开限定时间步长的 RNN 网络，一种称为截断时间反向传播的算法。

在 TensorFlow 中通过截断输入序列来简单实现这种功能。例如在时间序列预测问题上可以在训练时减小 `n_steps` 来实现截断。理所当然这种方法会限制模型在长期模式的学习能力。一种变通方案时确保缩短的序列中包含旧数据和新数据，从而使模型获得两者信息（如序列同时包含最近五个月的数据，最近五周的和最近五天的数据）。

问题时如何确保从去年的细分类中获取的数据有效性呢？这期间短暂但重要的事件对后世的影响，甚至时数年后这种影响是否一定要考虑在内呢（如选举结果）？这种方案有其先天的不足之处。

在长的时间训练过程中，第二个要面临的问题时第一个输入的记忆会在长时间运行的 RNN 网络中逐渐淡去。确实，通过变换的方式，数据穿流在 RNN 网络之中，每个时间步长后都有一些信息被抛弃掉了。那么在一定时间后，第一个输入实际上会在 RNN 的状态中消失于无形。

比如说，你想要分析长篇幅的影评的情感类别，影评以 `"I love this movie"` 开篇，并辅以各种改善影片的一些建议。试想一下，如果 RNN 网络逐渐忘记了开头的几个词，RNN 网络的判断完全有可能会对影评断章取义。

为了解决其中的问题，各种能够携带长时记忆的神经单元的变体被提出。这些变体是有效的，往往基本形式的神经单元就不怎么被使用了。

首先了解一下最流行的一种长时记忆神经单元：长短时记忆神经单元 LSTM。

LSTM 单元

长短时记忆单元在 1997 年由 S.H. 和 J.S. 首次提出 [3]，并在接下来的几年内经过 A.G，H.S [4]，W.Z [5] 等数位研究人员的改进逐渐形成。如果把 LSTM 单元看作一个黑盒，从外围看它和基本形式的记忆单元很相似，但 LSTM 单元会比基本单元性能更好，收敛更快，能够感知数据的长时依赖。TensorFlow 中通过 `BasicLSTMCell` 实现 LSTM 单元。

> [3]: "Long Short-Term Memory," S.Hochreiter and J.Schmidhuber(1997)
>
> [4]: "Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling," H.Sak et al.(2014)
>
> [5]: "Recurrent Neural Network Regularization," W.Zaremba et al.(2015)

```
lstm_cell = tf.contrib.rnn.BasicLSTMCell(num_units=n_neurons)
```

LSTM 单元的工作机制是什么呢？在图 14-13 中展示了基本 LSTM 单元的结构。

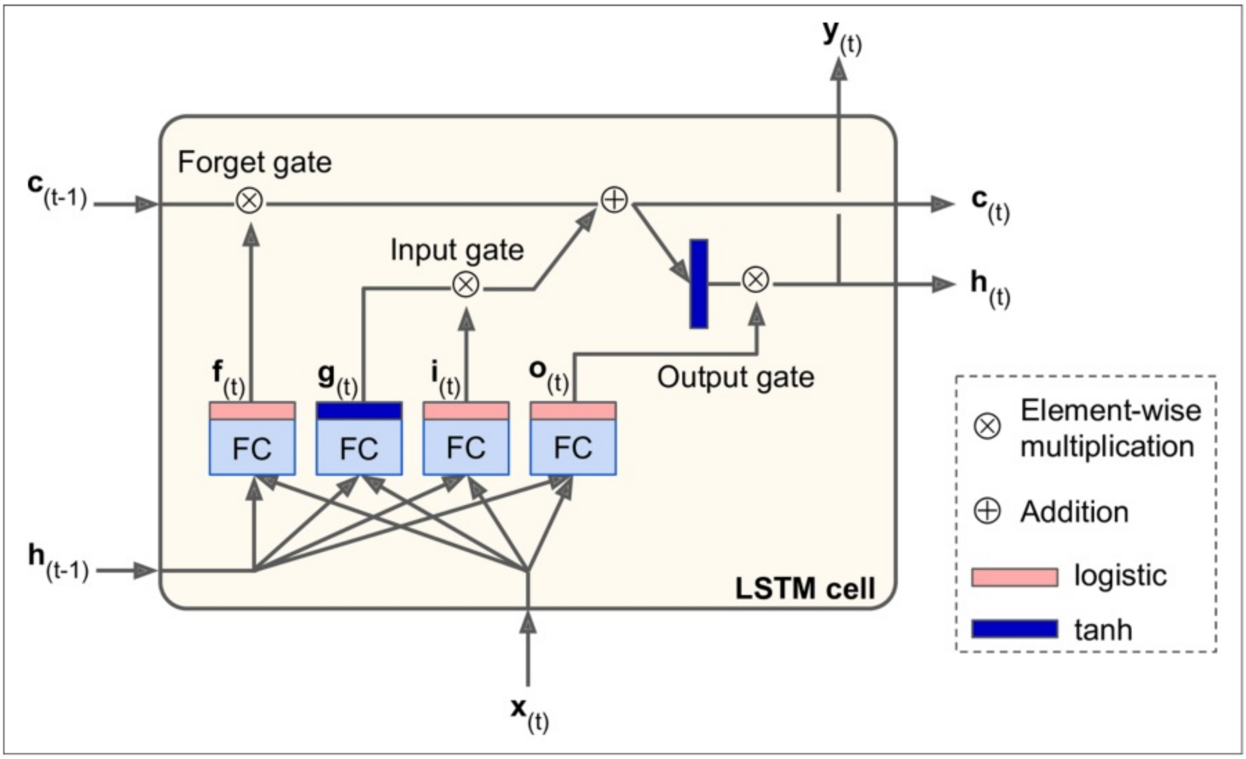


Figure 14-13. LSTM cell

不观察 LSTM 单元内部，除了一些不同外跟常规 RNN 单元极其相似。这些不同包括 LSTM 单元状态分为两个向量：$h^{(t)}$ 和 $c^{(t)}$（`c` 代表 cell）。可以简单认为 $h^{(t)}$ 是短期记忆状态，$c^{(t)}$ 是长期记忆状态。

好，我们来打开盒子。LSTM 单元的核心思想是其能够学习从长期状态中存储什么，忘记什么，读取什么。长期状态 $c^{(t-1)}$ 从左向右在网络中传播，依次经过遗忘门（forget gate）时丢弃一些记忆，之后加法操作增加一些记忆（从输入门中选择一些记忆）。输出 $c^{(t)}$ 不经任何转换直接输出。每个单位时间步长后，都有一些记忆被抛弃，新的记忆被添加进来。另一方面，长时状态经过 tanh 激活函数通过输出门得到短时记忆 $h^{(t)}$，同时它也是这一时刻的单元输出结果 $y^{(t)}$。接下来讨论一下新的记忆时如何产生的，门的功能是如何实现的。

首先，当前的输入向量 $x^{(t)}$ 和前一时刻的短时状态 $h^{(t-1)}$ 作为输入传给四个全连接层，这四个全连接层有不同的目的：

- 其中主要的全连接层输出 $g^{(t)}$，它的常规任务就是解析当前的输入 $x^{(t)}$ 和前一时刻的短时状态 $h^{(t-1)}$。在基本形式的 RNN 单元中，就与这种形式一样，直接输出了 $h^{(t)}$ 和 $y^{(t)}$。与之不同的是 LSTM 单元会将一部分 $g^{(t)}$ 存储在长时状态中。
- 其它三个全连接层被称为门控制器（gate controller）。其采用 Logistic 作为激活函数，输出范围在 0 到 1 之间。正如在结构图中所示，这三个层的输出提供给了逐元素乘法操作，当输入为 0 时门关闭，输出为 1 时门打开。分别为：
    - 遗忘门（forget gat）由 $f^{(t)}$ 控制，来决定哪些长期记忆需要被擦除；
    - 输入门（input gate） 由 $i^{(t)}$ 控制，它的作用是处理哪部分 $g^{(t)}$ 应该被添加到长时状态中，也就是为什么被称为部分存储。

- 输出门（output gate）由 $o^{(t)}$ 控制，在这一时刻的输出 $h^{(t)}$ 和 $y^{(t)}$ 就是由输出门控制的，从长时状态中读取的记忆。

简要来说，LSTM 单元能够学习到识别重要输入（输入门作用），存储进长时状态，并保存必要的时间（遗忘门功能），并学会提取当前输出所需要的记忆。

这也解释了 LSTM 单元能够在提取长时序列，长文本，录音等数据中的长期模式的惊人成功的原因。

公式 14-3 总结了如何计算单元的长时状态，短时状态，和单个输入情形时每单位步长的输出（小批量的方程形式与单输入的形式相似）。

Equation 14-3. LSTM computations

$$
\begin{aligned}
\mathbf{i}_{(t)} &= \sigma\left(\mathbf{W}_{xi}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hi}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_i\right) \\
\mathbf{f}_{(t)} &= \sigma\left(\mathbf{W}_{xf}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hf}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_f\right) \\
\mathbf{o}_{(t)} &= \sigma\left(\mathbf{W}_{xo}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{ho}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_o\right) \\
\mathbf{g}_{(t)} &= \tanh\left(\mathbf{W}_{xg}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_g\right) \\
\mathbf{c}_{(t)} &= \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \mathbf{g}_{(t)} \\
\mathbf{y}_{(t)} &= \mathbf{h}_{(t)} = \mathbf{o}_{(t)} \otimes \tanh\left(\mathbf{c}_{(t)}\right)
\end{aligned}
$$

- W_{xi}，W_{xf}，W_{xo}，W_{xg} 是四个全连接层关于输入向量 $x^{(t)}$ 的权重。
- W_{hi}，W_{hf}，W_{ho}，W_{hg} 是四个全连接层关于上一时刻的短时状态 $h^{(t-1)}$ 的权重。
- b_i，b_f，b_o，b_g 是全连接层的四个偏置项，需要注意的是 TensorFlow 将其初始化为全 1 向量，而非全 0，为了阻止网络初始训练状态下，各个门关闭从而忘记所有记忆。

窥孔连接

基本形式的 LSTM 单元中，门的控制仅有当前的输入 $x^{(t)}$ 和前一时刻的短时状态 $h^{(t-1)}$。不妨让各个控制门窥视一下长时状态，获取一些上下文信息不失为一种尝试。该想法由 F.G.he J.S. 在 2000 年提出。他们提出的 LSTM 的变体拥有叫做窥孔连接的额外连接：把前一时刻的长时状态 $c^{(t-1)}$ 加入遗忘门和输入门控制的输入，当前时刻的长时状态加入输出门的控制输入。

TensorFLow 中由 `LSTMCell` 实现以上变体 LSTM，并设置 `use_peepholes=True` 。

```
lstm_cell = tf.contrib.rnn.LSTMCell(num_units=n_neurons, use_peepholes=True)
```

在众多 LSTM 变体中，一个特别流行的变体就是 GRU 单元。

GRU 单元

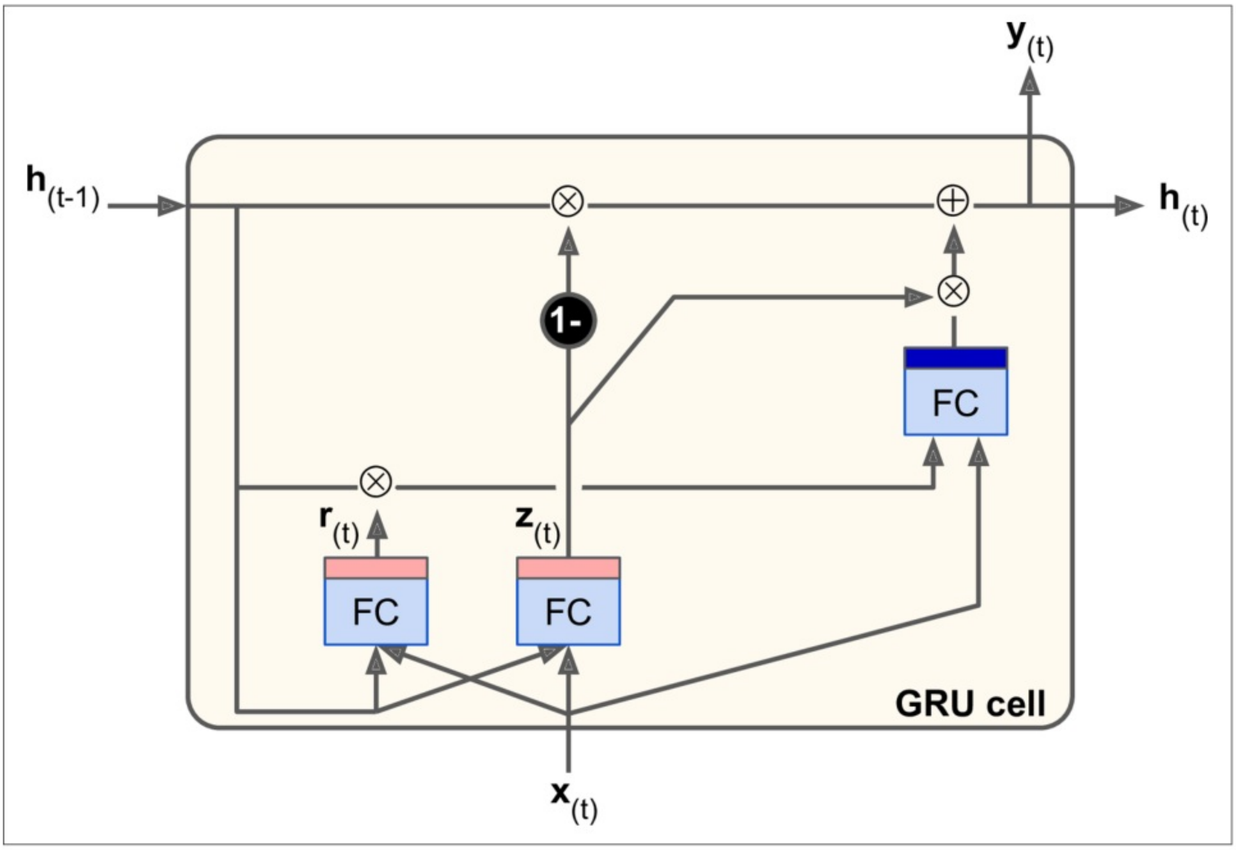


Figure 14-14. GRU cell

门控循环单元（图 14-14）在 2014 年的 K.Cho et al. 的论文中提出，并且此文也引入了前文所述的编解码网络。

门控循环单元是 LSTM 单元的简化版本，能实现同样的性能，这也说明了为什么它能越来越流行。简化主要在一下几个方面：

- 长时状态和短时状态合并为一个向量 $h^{(t)}$。
- 用同一个门控制遗忘门和输入门。如果门控制输入 1，输入门打开，遗忘门关闭，反之亦然。也就是说，如果当有新的记忆需要存储，那么就必须实现在其对应位置事先擦除该处记忆。这也构成了 LSTM 本身的常见变体。
- GRU 单元取消了输出门，单元的全部状态就是该时刻的单元输出。与此同时，增加了一个控制门 $r^{(t)}$ 来控制哪部分前一时间步的状态在该时刻的单元内呈现。

Equation 14-4. GRU computations

$$
\begin{aligned}
\mathbf{z}_{(t)} &= \sigma\left(\mathbf{W}_{xz}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hz}{}^{T} \cdot \mathbf{h}_{(t-1)}\right) \\
\mathbf{r}_{(t)} &= \sigma\left(\mathbf{W}_{xr}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hr}{}^{T} \cdot \mathbf{h}_{(t-1)}\right) \\
\mathbf{g}_{(t)} &= \tanh\left(\mathbf{W}_{xg}{}^{T} \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}{}^{T} \cdot \left(\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}\right)\right) \\
\mathbf{h}_{(t)} &= \left(1 - \mathbf{z}_{(t)}\right) \otimes \tanh\left(\mathbf{W}_{xg}{}^{T} \cdot \mathbf{h}_{(t-1)} + \mathbf{z}_{(t)} \otimes \mathbf{g}_{t}\right)
\end{aligned}
$$

公式 14-4 总结了如何计算单个输入情形时每单位步的单元的状态。

在 TensoFlow 中创建 GRU 单元很简单：

```
gru_cell = tf.contrib.rnn.GRUCell(n_units=n_neurons)
```

LSTM 或 GRU 单元是近年来 RNN 成功背后的主要原因之一，特别是在自然语言处理（NLP）中的应用。

自然语言处理

现在，大多数最先进的 NLP 应用（如机器翻译，自动摘要，解析，情感分析等），现在（至少一部分）都基于 RNN。 在最后一节中，我们将快速了解机器翻译模型的概况。 TensorFlow 的很厉害的 Word2Vec 和 Seq2Seq 教程非常好地介绍了这个主题，所以你一定要阅读一下。

单词嵌入

在我们开始之前，我们需要选择一个词的表示形式。 一种选择可以是，使用单热向量表示每个词。 假设你的词汇表包含 5 万个单词，那么第 `n` 个单词将被表示为 50,000 维的向量，除了第 `n` 个位置为 1 之外，其它全部为 0。 然而，对于如此庞大的词汇表，这种稀疏表示根本就不会有效。 理想情况下，你希望相似的单词具有相似的表示形式，这使得模型可以轻松地将所学的关于单词的只是，推广到所有相似单词。 例如，如果模型被告知 `"I drink milk"` 是一个有效的句子，并且如果它知道 `"milk"` 接近于 `"water"` ，而不同于 `"shoes"` ，那么它会知道 `"I drink water"` 也许是一个有效的句子，而 `"I drink shoes"` 可能不是。 但你如何提出这样一个有意义的表示呢？

最常见的解决方案是，用一个相当小且密集的向量（例如 150 维）表示词汇表中的每个单词，称为嵌入，并让神经网络在训练过程中，为每个单词学习一个良好的嵌入。 在训练开始时，嵌入只是随机选择的，但在训练过程中，反向传播会自动更新嵌入，来帮助神经网络执行任务。 通常这意味着，相似的词会逐渐彼此靠近，甚至最终以一种相当有意义的方式组织起来。 例如，嵌入可能最终沿着各种轴分布，它们代表性别，单数/复数，形容词/名词。 结果可能真的很神奇。

在TensorFlow中，首先需要创建一个变量来表示词汇表中每个词的嵌入（随机初始化）：

```
vocabulary_size = 50000
embedding_size = 150
embeddings = tf.Variable(
    tf.random_uniform([vocabulary_size, embedding_size], -1.0, 1.0))
```

现在假设你打算将句子 `"I drink milk"` 提供给你的神经网络。 你应该首先对句子进行预处理并将其分解成已知单词的列表。 例如，你可以删除不必要的字符，用预定义的标记词（如 `"[UNK]"` ）替换未知单词，用 `"[NUM]"` 替换数字值，用 `"[URL]"` 替换 URL 等。 一旦你有了一个已知单词列表，你可以在字典中查找每个单词的整数标识符（从 0 到 49999），例如 `[72，3335，288]` 。 此时，你已准备好使用占位符将这些单词标识符提供给 TensorFlow，并应用 `embedding_lookup()` 函数来获取相应的嵌入：

```
train_inputs = tf.placeholder(tf.int32, shape=[None])  # from ids...
embed = tf.nn.embedding_lookup(embeddings, train_inputs)  # ...to embeddings
```

一旦你的模型习得了良好的词嵌入，它们实际上可以在任何 NLP 应用中高效复用：毕竟， `"milk"` 依然接近于 `"water"` ，而且不管你的应用是什么，它都不同于 `"shoes"` 。 实际上，你可能需要下载预训练的单词嵌入，而不是训练自己的单词嵌入。 就像复用预训练层（参见第 11 章）一样，你可以选择冻结预训练嵌入（例如，使用 `trainable=False` 创建嵌入变量），或者让反向传播为你的应用调整它们。 第一种选择将加速训练，但第二种选择可能会产生稍高的性能。

> 提示
>
> 对于表示可能拥有大量不同值的类别属性，嵌入也很有用，特别是当值之间存在复杂的相似性的时候。 例如，考虑职业，爱好，菜品，物种，品牌等。

你现在拥有了实现机器翻译系统所需的几乎所有的工具。 现在我们来看看它吧。

用于机器翻译的编解码器网络

让我们来看看简单的机器翻译模型，它将英语句子翻译成法语（参见图 14-15）。

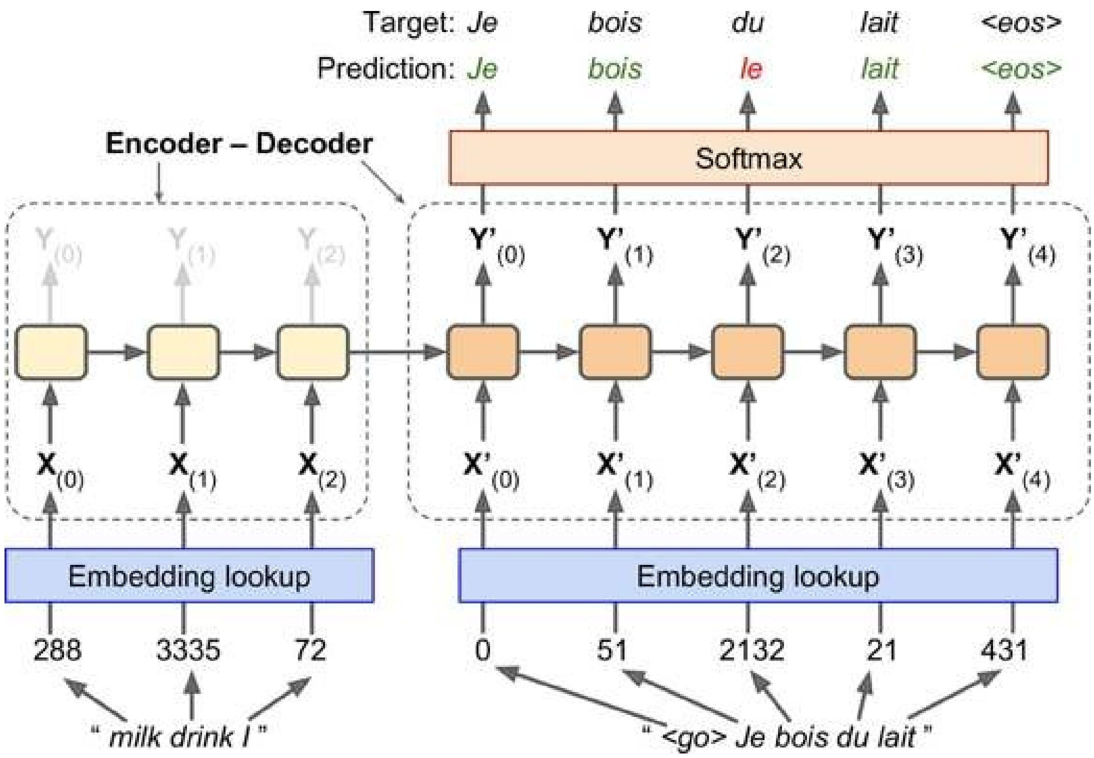


图 14-15：简单的机器翻译模型

英语句子被送进编码器，解码器输出法语翻译。 请注意，法语翻译也被用作解码器的输入，但后退了一步。 换句话说，解码器的输入是它应该在前一步输出的字（不管它实际输出的是什么）。 对于第一个单词，提供了表示句子开始的标记（ `"<go>"` ）。 解码器预期以序列末尾标记（EOS）结束句子（ `"<eos>"` ）。

请注意，英语句子在送入编码器之前会反转。 例如， `"I drink milk"` 与 `"milk drink I"` 相反。这确保了英语句子的开头将会最后送到编码器，这很有用，因为这通常是解码器需要翻译的第一个东西。

每个单词最初由简单整数标识符表示（例如，单词 `"milk"` 为 288）。 接下来，嵌入查找返回词的嵌入（如前所述，这是一个密集的，相当低维的向量）。 这些词的嵌入是实际送到编码器和解码器的内容。

在每个步骤中，解码器输出输出词汇表（即法语）中每个词的得分，然后 Softmax 层将这些得分转换为概率。 例如，在第一步中，单词 `"Je"` 有 20% 的概率， `"Tu"` 有 1% 的概率，以此类推。 概率最高的词会输出。 这非常类似于常规分类任务，因此你可以使用 `softmax_cross_entropy_with_logits()` 函数来训练模型。

请注意，在推断期间（训练之后），你不再将目标句子送入解码器。 相反，只需向解码器提供它在上一步输出的单词，如图 14-16 所示（这将需要嵌入查找，它未在图中显示）。

图 14-16：在推断期间，将之前的输出单词提供为输入

好的，现在你有了大方向。 但是，如果你阅读 TensorFlow 的序列教程，并查看 `rnn/translate/seq2seq_model.py` 中的代码（在 TensorFlow 模型中），你会注意到一些重要的区别：

- 首先，到目前为止，我们已经假定所有输入序列（编码器和解码器的）具有恒定的长度。但显然句子长度可能会有所不同。有几种方法可以处理它 - 例如，使用 `static_rnn()` 或 `dynamic_rnn()` 函数的 `sequence_length` 参数，来指定每个句子的长度（如前所述）。然而，教程中使用了另一种方法（大概是出于性能原因）：句子分到长度相似的桶中（例如，句子的单词 1 到 6 分到一个桶，单词 7 到 12 分到另一个桶，等等），并且使用特殊的填充标记（例如 `"<pad>"`）来填充较短的句子。例如，`"I drink milk"` 变成 `"<pad> <pad> <pad> milk drink I"`，翻译成 `"Je bois du lait <eos> <pad>"`。当然，我们希望忽略任何 EOS 标记之后的输出。为此，本教程的实现使用 `target_weights` 向量。例如，对于目标句子 `"Je bois du lait <eos> <pad>"`，权重将设置为 `[1.0,1.0,1.0,1.0,1.0,0.0]`（注意权重 0.0 对应目标句子中的填充标记）。简单地将损失乘以目标权重，将消除对应 EOS 标记之后的单词的损失。

- 其次，当输出词汇表很大时（就是这里的情况），输出每个可能的单词的概率将会非常慢。 如果目标词汇表包含 50,000 个法语单词，则解码器将输出 50,000 维向量，然后在这样的大向量上计算 softmax 函数，计算量将非常大。 为了避免这种情况，一种解决方案是让解码器输出更小的向量，例如 1,000 维向量，然后使用采样技术来估计损失，而不必对目标词汇表中的每个单词计算它。 这种采样 Softmax 技术是由 SébastienJean 等人在 2015 年提出的。在 TensorFlow 中，你可以使用 `sampled_softmax_loss()` 函数。

- 第三，教程的实现使用了一种注意力机制，让解码器能够窥视输入序列。 注意力增强的 RNN 不在本书的讨论范围之内，但如果你有兴趣，可以关注机器翻译，机器阅读和图像说明的相关论文。

- 最后，本教程的实现使用了 `tf.nn.legacy_seq2seq` 模块，该模块提供了轻松构建各种编解码器模型的工具。 例如，`embedding_rnn_seq2seq()` 函数会创建一个简单的编解码器模型，它会自动为你处理单词嵌入，就像图 14-15 中所示的一样。 此代码可能会很快更新，来使用新的 `tf.nn.seq2seq` 模块。

你现在拥有了，了解所有 seq2seq 教程的实现所需的全部工具。 将它们取出，并训练你自己的英法翻译器吧！

练习

1. 你能想象 seq2seq RNN 的几个应用吗？ seq2vec 的 RNN 呢？vex2seq 的 RNN 呢？
2. 为什么人们使用编解码器 RNN 而不是简单的 seq2seq RNN 来自动翻译？
3. 如何将卷积神经网络与 RNN 结合，来对视频进行分类？
4. 使用 `dynamic_rnn()` 而不是 `static_rnn()` 构建 RNN 有什么好处？
5. 你如何处理长度可变的输入序列？ 那么长度可变输出序列呢？
6. 在多个 GPU 上分配深层 RNN 的训练和执行的常见方式是什么？
7. Hochreiter 和 Schmidhuber 在其关于 LSTM 的文章中使用了嵌入式 Reber 语法。 它们是产生字符串，如 `"BPBTSXXVPSEPE"` 的人造语法。查看 Jenny Orr 对此主题的不错的介

绍。选择一个特定的嵌入式 Reber 语法（例如 Jenny Orr 页面上显示的语法），然后训练一个 RNN 来确定字符串是否遵循该语法。你首先需要编写一个函数，该函数能够生成训练批量，包含大约 50% 遵循语法的字符串，以及 50% 不遵循的字符串。

8. 解决“How much did it rain? II”（下雨下了多久 II）Kaggle 比赛。这是一个时间序列预测任务：它为你提供极化雷达值的快照，并要求预测每小时降水量。Luis Andre Dutra e Silva 的采访对他在比赛中获得第二名的技术，提供了一些有趣的见解。特别是，他使用了由两个 LSTM 层组成的 RNN。
9. 通过 TensorFlow 的 Word2Vec 教程来创建单词嵌入，然后通过 Seq2Seq 教程来训练英法翻译系统。

附录 A 提供了这些练习的答案。

第15章 自编码器

自编码器是能够在无监督的情况下学习输入数据的有效表示（叫做编码）的人工神经网络（即，训练集是未标记）。这些编码通常具有比输入数据低得多的维度，使得自编码器对降维有用（参见第 8 章）。更重要的是，自编码器可以作为强大的特征检测器，它们可以用于无监督的深度神经网络预训练（正如我们在第 11 章中讨论过的）。最后，他们能够随机生成与训练数据非常相似的新数据；这被称为生成模型。例如，您可以在脸部图片上训练自编码器，然后可以生成新脸部。

令人惊讶的是，自编码器只需学习将输入复制到其输出即可工作。 这听起来像是一件小事，但我们会看到以各种方式约束网络可能会让它变得相当困难。例如，您可以限制内部表示的大小，或者可以向输入添加噪声并训练网络以恢复原始输入。这些约束防止自编码器将输入直接复制到输出，这迫使它学习表示数据的有效方法。 简言之，编码是自编码器在某些限制条件下尝试学习恒等函数的副产品。

在本章中，我们将更深入地解释自编码器如何工作，可以施加什么类型的约束以及如何使用 TensorFlow 实现它们，无论是用来降维，特征提取，无监督预训练还是作为生成式模型。

有效的数据表示

您发现以下哪一个数字序列最容易记忆？

- `40, 27, 25, 36, 81, 57, 10, 73, 19, 68`
- `50, 25, 76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20`

乍一看，第一个序列似乎应该更容易，因为它要短得多。 但是，如果仔细观察第二个序列，则可能会注意到它遵循两条简单规则：偶数是前面数的一半，奇数是前面数的三倍加一（这是一个着名的序列，称为雹石序列）。一旦你注意到这种模式，第二个序列比第一个更容易记忆，因为你只需要记住两个规则，第一个数字和序列的长度。 请注意，如果您可以快速轻松地记住非常长的序列，则您不会在意第二个序列中存在的模式。 你只需要了解每一个数字，就是这样。 事实上，很难记住长序列，因此识别模式非常有用，并且希望能够澄清为什么在训练过程中限制自编码器会促使它发现并利用数据中的模式。

记忆，感知和模式匹配之间的关系在 20 世纪 70 年代早期由 William Chase 和 Herbert Simon 着名研究。 他们观察到，专家棋手能够通过观看棋盘5秒钟来记忆所有棋子的位置，这是大多数人认为不可能完成的任务。 然而，只有当这些棋子被放置在现实位置（来自实际比赛）时才是这种情况，而不是随机放置棋子。 国际象棋专家没有比你更好的记忆，他们只是更容易看到国际象棋模式，这要归功于他们对比赛的经验。 注意模式有助于他们有效地存储信息。

就像这个记忆实验中的象棋棋手一样，一个自编码器会查看输入信息，将它们转换为高效的内部表示形式，然后吐出一些（希望）看起来非常接近输入的东西。 自编码器总是由两部分组成：将输入转换为内部表示的编码器（或识别网络），然后是将内部表示转换为输出的解码器（或生成网络）（见图 15-1）。

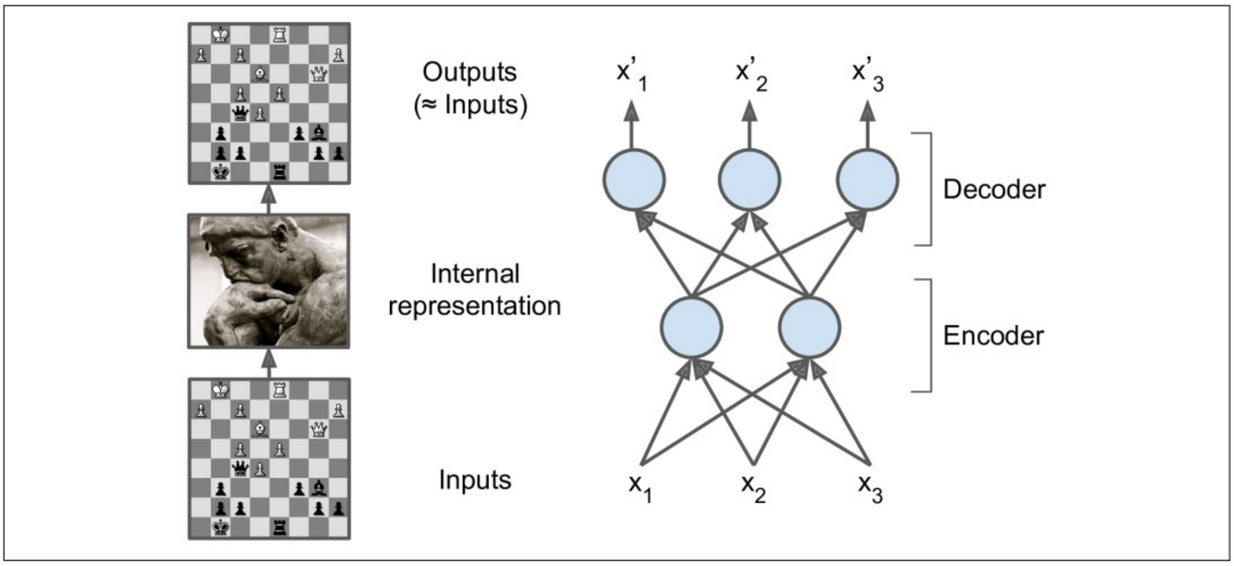


Figure 15-1. The chess memory experiment (left) and a simple autoencoder (right)

如您所见，自编码器通常具有与多层感知器（MLP，请参阅第 10 章）相同的体系结构，但输出层中的神经元数量必须等于输入数量。 在这个例子中，只有一个由两个神经元（编码器）组成的隐藏层和一个由三个神经元（解码器）组成的输出层。 由于自编码器试图重构输入，所以输出通常被称为重建，并且损失函数包含重建损失，当重建与输入不同时，重建损失会对模型进行惩罚。

由于内部表示具有比输入数据更低的维度（它是 2D 而不是 3D），所以自编码器被认为是不完整的。 不完整的自编码器不能简单地将其输入复制到编码，但它必须找到一种方法来输出其输入的副本。 它被迫学习输入数据中最重要的特征（并删除不重要的特征）。

我们来看看如何实现一个非常简单的不完整的自编码器，以降低维度。

用不完整的线性自编码器执行 PCA

如果自编码器仅使用线性激活并且损失函数是均方误差（MSE），则可以显示它最终执行主成分分析（参见第 8 章）。

以下代码构建了一个简单的线性自编码器，以在 3D 数据集上执行 PCA，并将其投影到 2D：

```
import tensorflow as tf
from tensorflow.contrib.layers import fully_connected
n_inputs = 3 # 3D inputs
n_hidden = 2 # 2D codings
n_outputs = n_inputs
learning_rate = 0.01
X = tf.placeholder(tf.float32, shape=[None, n_inputs])
hidden = fully_connected(X, n_hidden, activation_fn=None)
outputs = fully_connected(hidden, n_outputs, activation_fn=None)
reconstruction_loss = tf.reduce_mean(tf.square(outputs - X)) # MSE
optimizer = tf.train.AdamOptimizer(learning_rate)
training_op = optimizer.minimize(reconstruction_loss)
init = tf.global_variables_initializer()
```

这段代码与我们在过去章节中建立的所有 MLP 没有什么不同。 需要注意的两件事是：

- 输出的数量等于输入的数量。
- 为了执行简单的 PCA，我们设置 `activation_fn = None` （即，所有神经元都是线性的）

而损失函数是 MSE。 我们很快会看到更复杂的自编码器。

现在让我们加载数据集，在训练集上训练模型，并使用它来对测试集进行编码（即将其投影到 2D）：

```
X_train, X_test = [...] # load the dataset
n_iterations = 1000
codings = hidden # the output of the hidden layer provides the codings
with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        training_op.run(feed_dict={X: X_train}) # no labels (unsupervised)
    codings_val = codings.eval(feed_dict={X: X_test})
```

图 15-2 显示了原始 3D 数据集（左侧）和自编码器隐藏层的输出（即编码层，右侧）。 正如您所看到的，自编码器找到了将数据投影到数据上的最佳二维平面，保留了数据的尽可能多的差异（就像 PCA 一样）。

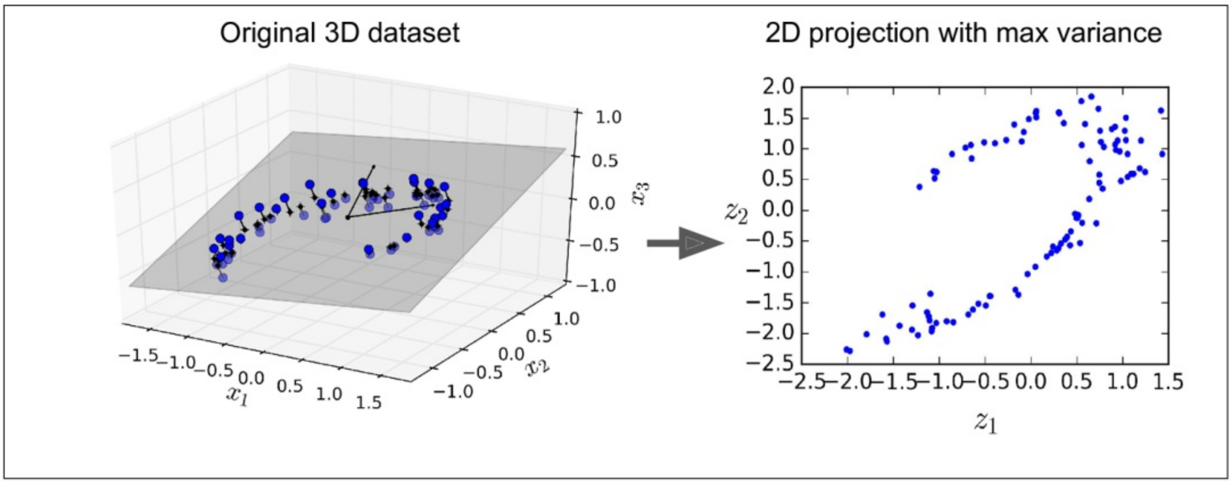


Figure 15-2. PCA performed by an undercomplete linear autoencoder

栈式自编码器（SAE）

就像我们讨论过的其他神经网络一样，自编码器可以有多个隐藏层。 在这种情况下，它们被称为栈式自编码器（或深度自编码器）。 添加更多层有助于自编码器了解更复杂的编码。 但是，必须注意不要让自编码器功能太强大。 设想一个编码器非常强大，只需学习将每个输入映射到一个任意数字（并且解码器学习反向映射）即可。 很明显，这样的自编码器将完美地重构训练数据，但它不会在过程中学习到任何有用的数据表示（并且它不可能很好地推广到新的实例）。

栈式自编码器的架构关于中央隐藏层（编码层）通常是对称的。 简单来说，它看起来像一个三明治。 例如，一个用于 MNIST 的自编码器（在第 3 章中介绍）可能有 784 个输入，其次是一个隐藏层，有 300 个神经元，然后是一个中央隐藏层，有 150 个神经元，然后是另一个隐藏层，有 300 个神经元，输出层有 784 神经元。 这个栈式自编码器如图 15-3 所示。

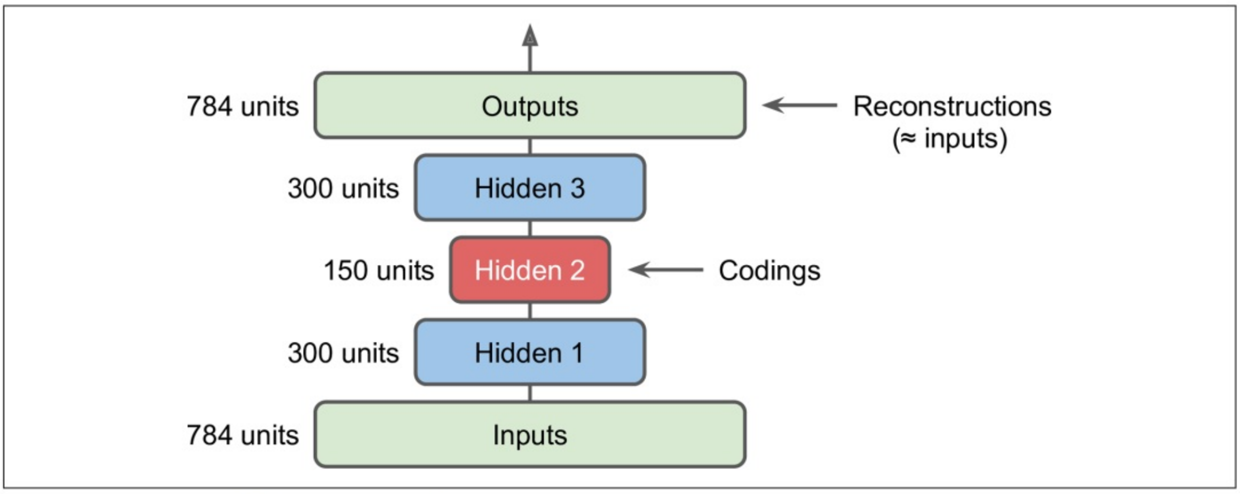


Figure 15-3. Stacked autoencoder

TensorFlow实现

您可以像常规深度 MLP 一样实现栈式自编码器。 特别是，我们在第 11 章中用于训练深度网络的技术也可以应用。例如，下面的代码使用 He 初始化，ELU 激活函数和 l2 正则化为 MNIST 构建一个栈式自编码器。 代码应该看起来很熟悉，除了没有标签（没有 `y`）：

```
n_inputs = 28 * 28 # for MNIST
n_hidden1 = 300
n_hidden2 = 150 # codings
n_hidden3 = n_hidden1
n_outputs = n_inputs

learning_rate = 0.01
l2_reg = 0.001

X = tf.placeholder(tf.float32, shape=[None, n_inputs])
with tf.contrib.framework.arg_scope(
        [fully_connected],
        activation_fn=tf.nn.elu,
        weights_initializer=tf.contrib.layers.variance_scaling_initializer(),
        weights_regularizer=tf.contrib.layers.l2_regularizer(l2_reg)):
    hidden1 = fully_connected(X, n_hidden1)
    hidden2 = fully_connected(hidden1, n_hidden2) # codings
    hidden3 = fully_connected(hidden2, n_hidden3)
    outputs = fully_connected(hidden3, n_outputs, activation_fn=None)
reconstruction_loss = tf.reduce_mean(tf.square(outputs - X)) # MSE

reg_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
loss = tf.add_n([reconstruction_loss] + reg_losses)

optimizer = tf.train.AdamOptimizer(learning_rate)
training_op = optimizer.minimize(loss)

init = tf.global_variables_initializer()
```

然后可以正常训练模型。请注意，数字标签（`y_batch`）未使用：

```
n_epochs = 5
batch_size = 150
with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        n_batches = mnist.train.num_examples // batch_size
        for iteration in range(n_batches):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch})
```

关联权重

当自编码器整齐地对称时，就像我们刚刚构建的那样，一种常用技术是将解码器层的权重与编码器层的权重相关联。这样减少了模型中的权重数量，加快了训练速度，并限制了过度拟合的风险。

具体来说，如果自编码器总共具有 `N` 个层（不计入输入层），并且 $W^{[L]}$ 表示第 `L` 层的连接权重（例如，层 1 是第一隐藏层，则层 `N / 2` 是编码 层，而层 `N` 是输出层），则解码器层权重可以简单地定义为：$W^{[N-L+1]} = W^{[L]T}$（其中 `L = 1, 2, ..., N2`）。

不幸的是，使用 `fully_connected()` 函数在 TensorFlow 中实现相关权重有点麻烦；手动定义层实际上更容易。代码结尾明显更加冗长：

```
activation = tf.nn.elu
regularizer = tf.contrib.layers.l2_regularizer(l2_reg)
initializer = tf.contrib.layers.variance_scaling_initializer()

X = tf.placeholder(tf.float32, shape=[None, n_inputs])

weights1_init = initializer([n_inputs, n_hidden1])
weights2_init = initializer([n_hidden1, n_hidden2])

weights1 = tf.Variable(weights1_init, dtype=tf.float32, name="weights1")
weights2 = tf.Variable(weights2_init, dtype=tf.float32, name="weights2")
weights3 = tf.transpose(weights2, name="weights3") # tied weights
weights4 = tf.transpose(weights1, name="weights4") # tied weights

biases1 = tf.Variable(tf.zeros(n_hidden1), name="biases1")
biases2 = tf.Variable(tf.zeros(n_hidden2), name="biases2")
biases3 = tf.Variable(tf.zeros(n_hidden3), name="biases3")
biases4 = tf.Variable(tf.zeros(n_outputs), name="biases4")

hidden1 = activation(tf.matmul(X, weights1) + biases1)
hidden2 = activation(tf.matmul(hidden1, weights2) + biases2)
hidden3 = activation(tf.matmul(hidden2, weights3) + biases3)
outputs = tf.matmul(hidden3, weights4) + biases4

reconstruction_loss = tf.reduce_mean(tf.square(outputs - X))
reg_loss = regularizer(weights1) + regularizer(weights2)
loss = reconstruction_loss + reg_loss

optimizer = tf.train.AdamOptimizer(learning_rate)
training_op = optimizer.minimize(loss)

init = tf.global_variables_initializer()
```

这段代码非常简单，但有几件重要的事情需要注意：

- 首先，权重 3 和权重 4 不是变量，它们分别是权重 2 和权重 1 的转置（它们与它们“绑定”）。
- 其次，由于它们不是变量，所以规范它们是没有用的：我们只调整权重 1 和权重 2。
- 第三，偏置永远不会被束缚，并且永远不会正规化。

一次训练一个自编码器

我们不是一次完成整个栈式自编码器的训练，而是一次训练一个浅自编码器，然后将所有这些自编码器堆叠到一个栈式自编码器（因此名称）中，通常要快得多，如图 15-4 所示。 这对于非常深的自编码器特别有用。

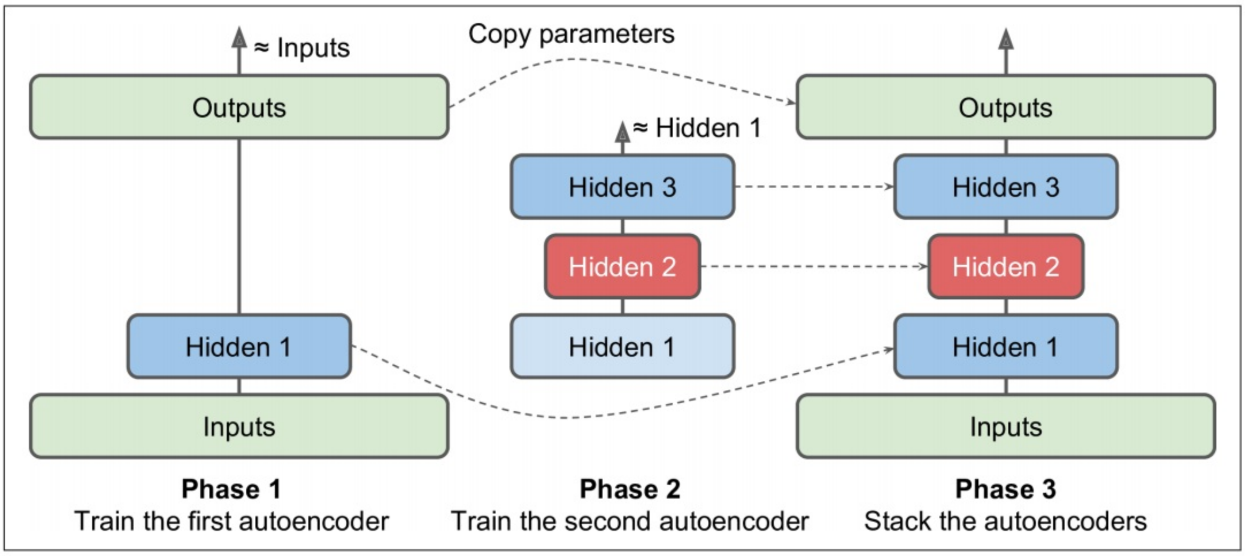


Figure 15-4. Training one autoencoder at a time

在训练的第一阶段，第一个自编码器学习重构输入。 在第二阶段，第二个自编码器学习重构第一个自编码器隐藏层的输出。 最后，您只需使用所有这些自编码器来构建一个大三明治，如图 15-4 所示（即，您首先将每个自编码器的隐藏层，然后按相反顺序堆叠输出层）。 这给你最后的栈式自编码器。 您可以用这种方式轻松地训练更多的自编码器，构建一个非常深的栈式自编码器。

为了实现这种多阶段训练算法，最简单的方法是对每个阶段使用不同的 TensorFlow 图。 训练完一个自编码器后，您只需通过它运行训练集并捕获隐藏层的输出。 这个输出作为下一个自编码器的训练集。 一旦所有自编码器都以这种方式进行了训练,您只需复制每个自编码器的权重和偏置，然后使用它们来构建堆叠的自编码器。 实现这种方法非常简单，所以我们不在这里详细说明，但请查阅 Jupyter notebooks 中的代码作为示例。

另一种方法是使用包含整个栈式自编码器的单个图，以及执行每个训练阶段的一些额外操作，如图 15-5 所示。

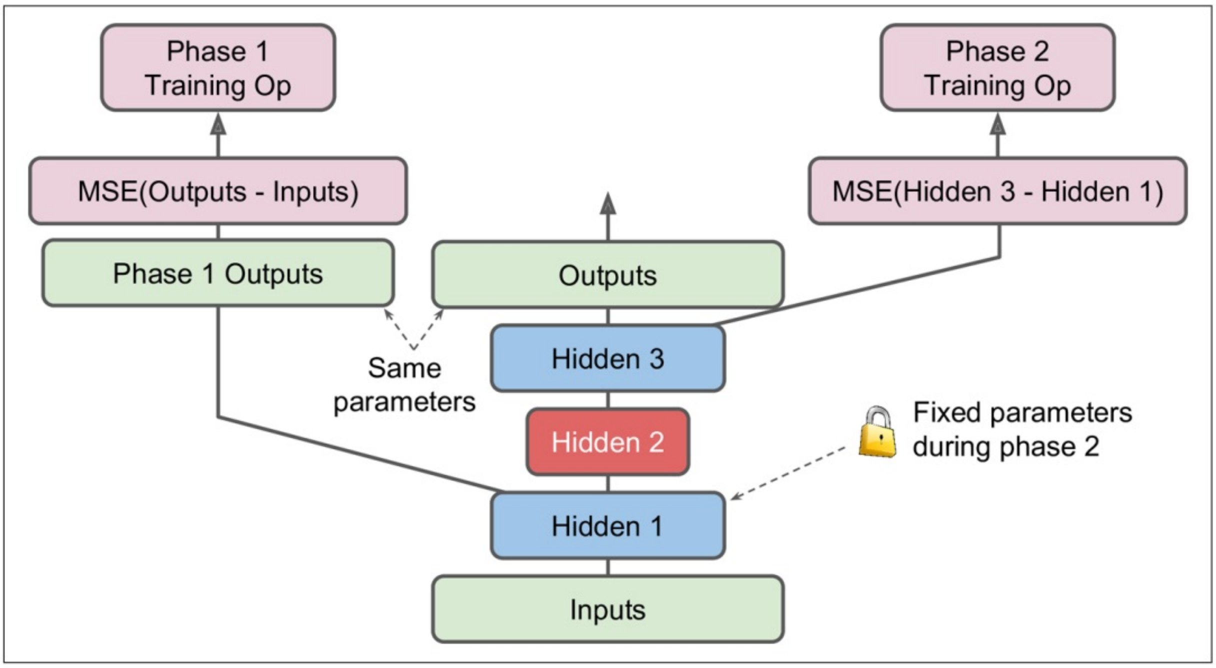


Figure 15-5. A single graph to train a stacked autoencoder

这值得解释一下：

- 图中的中央列是完整的栈式自编码器。这部分可以在训练后使用。
- 左列是运行第一阶段训练所需的一系列操作。它创建一个绕过隐藏层 2 和 3 的输出层。该输出层与堆叠的自编码器的输出层共享相同的权重和偏置。此外还有旨在使输出尽可能接近输入的训练操作。因此，该阶段将训练隐藏层1和输出层（即，第一自编码器）的权重和偏置。
- 图中的右列是运行第二阶段训练所需的一组操作。它增加了训练操作，目的是使隐藏层 3 的输出尽可能接近隐藏层 1 的输出。注意，我们必须在运行阶段 2 时冻结隐藏层 1。此阶段将训练隐藏层 1 的权重和偏置给隐藏的层 2 和 3（即第二自 编码器）。

TensorFlow 代码如下所示：

```
[...] # Build the whole stacked autoencoder normally.
# In this example, the weights are not tied.
optimizer = tf.train.AdamOptimizer(learning_rate)

with tf.name_scope("phase1"):
    phase1_outputs = tf.matmul(hidden1, weights4) + biases4
    phase1_reconstruction_loss = tf.reduce_mean(tf.square(phase1_outputs - X))
    phase1_reg_loss = regularizer(weights1) + regularizer(weights4)
    phase1_loss = phase1_reconstruction_loss + phase1_reg_loss
    phase1_training_op = optimizer.minimize(phase1_loss)

with tf.name_scope("phase2"):
    phase2_reconstruction_loss = tf.reduce_mean(tf.square(hidden3 - hidden1))
    phase2_reg_loss = regularizer(weights2) + regularizer(weights3)
    phase2_loss = phase2_reconstruction_loss + phase2_reg_loss
    train_vars = [weights2, biases2, weights3, biases3]
    phase2_training_op = optimizer.minimize(phase2_loss, var_list=train_vars)
```

第一阶段比较简单：我们只创建一个跳过隐藏层 2 和 3 的输出层，然后构建训练操作以最小化输出和输入之间的距离（加上一些正则化）。

第二阶段只是增加了将隐藏层 3 和隐藏层 1 的输出之间的距离最小化的操作（还有一些正则化）。 最重要的是，我们向 `minim()` 方法提供可训练变量的列表，确保省略权重 1 和偏差 1；这有效地冻结了阶段 2 期间的隐藏层 1。

在执行阶段，你需要做的就是为阶段 1 一些迭代进行训练操作，然后阶段 2 训练运行更多的迭代。

由于隐藏层 1 在阶段 2 期间被冻结，所以对于任何给定的训练实例其输出将总是相同的。 为了避免在每个时期重新计算隐藏层1的输出，您可以在阶段 1 结束时为整个训练集计算它，然后直接在阶段 2 中输入隐藏层 1 的缓存输出。这可以得到一个不错的性能上的提升。

可视化重建

确保自编码器得到适当训练的一种方法是比较输入和输出。 它们必须非常相似，差异应该是不重要的细节。 我们来绘制两个随机数字及其重建：

```
n_test_digits = 2
X_test = mnist.test.images[:n_test_digits]

with tf.Session() as sess:
    [...] # Train the Autoencoder
    outputs_val = outputs.eval(feed_dict={X: X_test})

def plot_image(image, shape=[28, 28]):
    plt.imshow(image.reshape(shape), cmap="Greys", interpolation="nearest")
    plt.axis("off")

for digit_index in range(n_test_digits):
    plt.subplot(n_test_digits, 2, digit_index * 2 + 1)
    plot_image(X_test[digit_index])
    plt.subplot(n_test_digits, 2, digit_index * 2 + 2)
    plot_image(outputs_val[digit_index])
```

Figure 15-6 shows the resulting images.

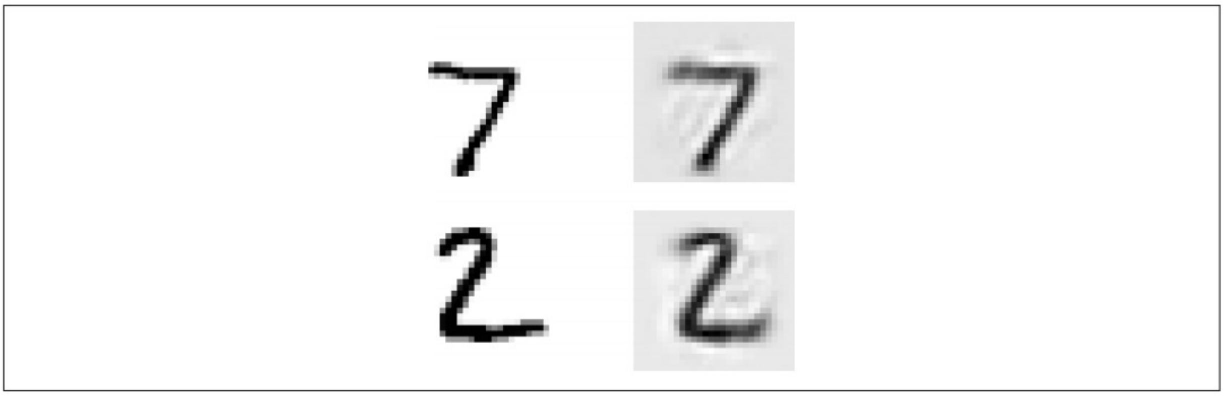


Figure 15-6. Original digits (left) and their reconstructions (right)

看起来够接近。 所以自编码器已经适当地学会了重现它，但是它学到了有用的特性？ 让我们来看看。

可视化功能

一旦你的自编码器学习了一些功能，你可能想看看它们。 有各种各样的技术。 可以说最简单的技术是在每个隐藏层中考虑每个神经元，并找到最能激活它的训练实例。 这对顶层隐藏层特别有用，因为它们通常会捕获相对较大的功能，您可以在包含它们的一组训练实例中轻松找到这些功能。 例如，如果神经元在图片中看到一只猫时强烈激活，那么激活它的图片最显眼的地方都会包含猫。 然而，对于较低层，这种技术并不能很好地工作，因为这些特征更小，更抽象，因此很难准确理解神经元正在为什么而兴奋。

让我们看看另一种技术。 对于第一个隐藏层中的每个神经元，您可以创建一个图像，其中像素的强度对应于给定神经元的连接权重。 例如，以下代码绘制了第一个隐藏层中五个神经元学习的特征：

```
with tf.Session() as sess:
    [...] # train autoencoder
    weights1_val = weights1.eval()

for i in range(5):
    plt.subplot(1, 5, i + 1)
    plot_image(weights1_val.T[i])
```

您可能会得到如图 15-7 所示的低级功能。

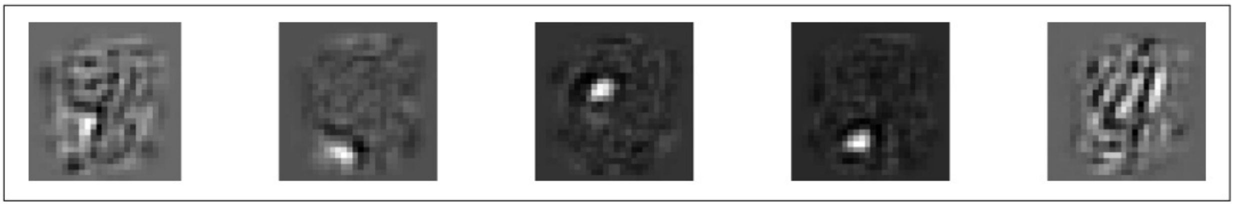


Figure 15-7. Features learned by five neurons from the first hidden layer

前四个特征似乎对应于小块，而第五个特征似乎寻找垂直笔划（请注意，这些特征来自堆叠去噪自编码器，我们将在后面讨论）。

另一种技术是给自编码器提供一个随机输入图像，测量您感兴趣的神经元的激活，然后执行反向传播来调整图像，使神经元激活得更多。 如果迭代数次（执行渐变上升），图像将逐渐变成最令人兴奋的图像（用于神经元）。 这是一种有用的技术，用于可视化神经元正在寻找的输入类型。

最后，如果使用自编码器执行无监督预训练（例如，对于分类任务），验证自编码器学习的特征是否有用的一种简单方法是测量分类器的性能。

无监督预训练使用栈式自编码器

正如我们在第 11 章中讨论的那样，如果您正在处理复杂的监督任务，但您没有大量标记的训练数据，则一种解决方案是找到执行类似任务的神经网络，然后重新使用其较低层。 这样就可以仅使用很少的训练数据来训练高性能模型，因为您的神经网络不必学习所有的低级特征；它将重新使用现有网络学习的特征检测器。

同样，如果您有一个大型数据集，但大多数数据集未标记，您可以先使用所有数据训练栈式自编码器，然后重新使用较低层为实际任务创建一个神经网络，并使用标记数据对其进行训练。 例如，图 15-8 显示了如何使用栈式自编码器为分类神经网络执行无监督预训练。 正如前面讨论过的，栈式自编码器本身通常每次都会训练一个自编码器。 在训练分类器时，如果您确实没有太多标记的训练数据，则可能需要冻结预训练层（至少是较低层）。

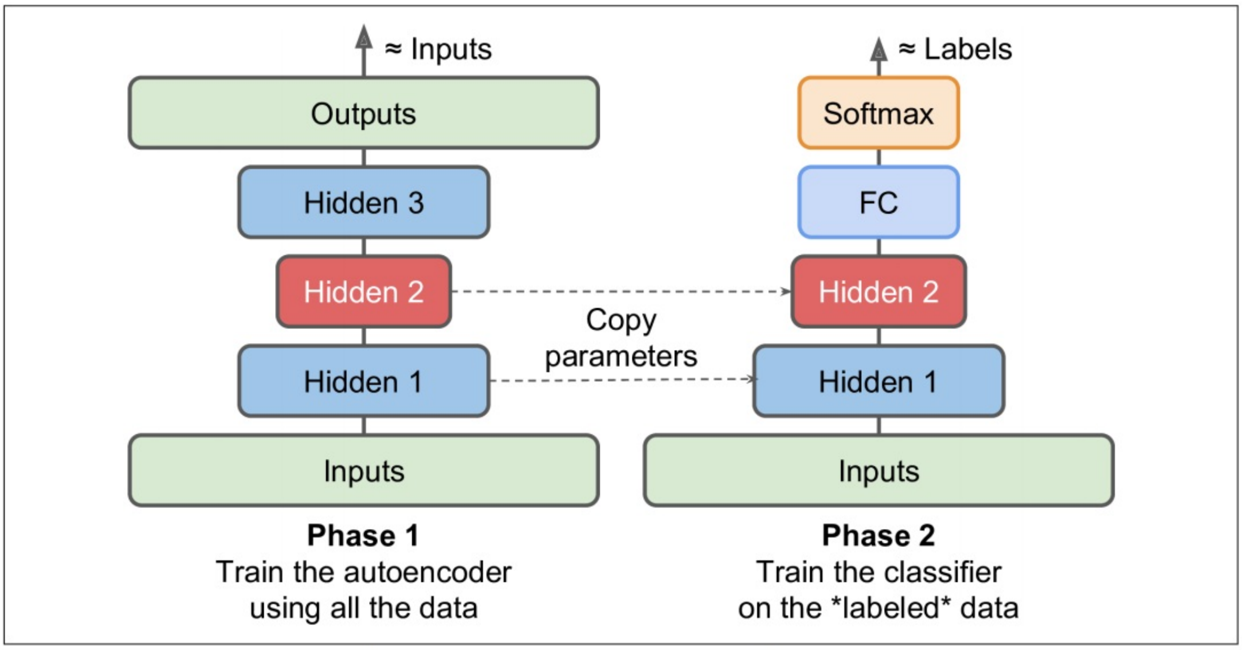


Figure 15-8. Unsupervised pretraining using autoencoders

这种情况实际上很常见，因为构建一个大型的无标签数据集通常很便宜（例如，一个简单的脚本可以从互联网上下载数百万张图像），但只能由人类可靠地标记它们（例如，将图像分类为可爱或不可爱）。 标记实例是耗时且昂贵的，因此只有几千个标记实例是很常见的。

正如我们前面所讨论的那样，当前深度学习海啸的触发因素之一是 Geoffrey Hinton 等人在 2006 年的发现，深度神经网络可以以无监督的方式进行预训练。 他们使用受限玻尔兹曼机器（见附录 E），但在 2007 年 Yoshua Bengio 等人表明自编码器也起作用。

TensorFlow 的实现没有什么特别之处：只需使用所有训练数据训练自编码器，然后重用其编码器层以创建一个新的神经网络（有关如何重用预训练层的更多详细信息，请参阅第 11 章或查看 Jupyte notebooks 中的代码示例）。

到目前为止，为了强制自编码器学习有趣的特性，我们限制了编码层的大小，使其不够完善。 实际上可以使用许多其他类型的约束，包括允许编码层与输入一样大或甚至更大的约束，导致过度完成的自编码器。 现在我们来看看其中的一些方法。

降噪自编码（DAE）

另一种强制自编码器学习有用功能的方法是为其输入添加噪声，对其进行训练以恢复原始的无噪声输入。 这可以防止自编码器将其输入复制到其输出，因此最终不得不在数据中查找模式。

自 20 世纪 80 年代以来，使用自编码器消除噪音的想法已经出现（例如，在 Yann LeCun 的 1987 年硕士论文中提到过）。 在 2008 年的一篇论文中，帕斯卡尔文森特等人。 表明自编码器也可用于特征提取。 在 2010 年的一篇文章中 Vincent 等人引入堆叠降噪自编码器。

噪声可以是纯粹的高斯噪声添加到输入，或者它可以随机关闭输入，就像 drop out（在第 11 章介绍）。 图 15-9 显示了这两个选项。

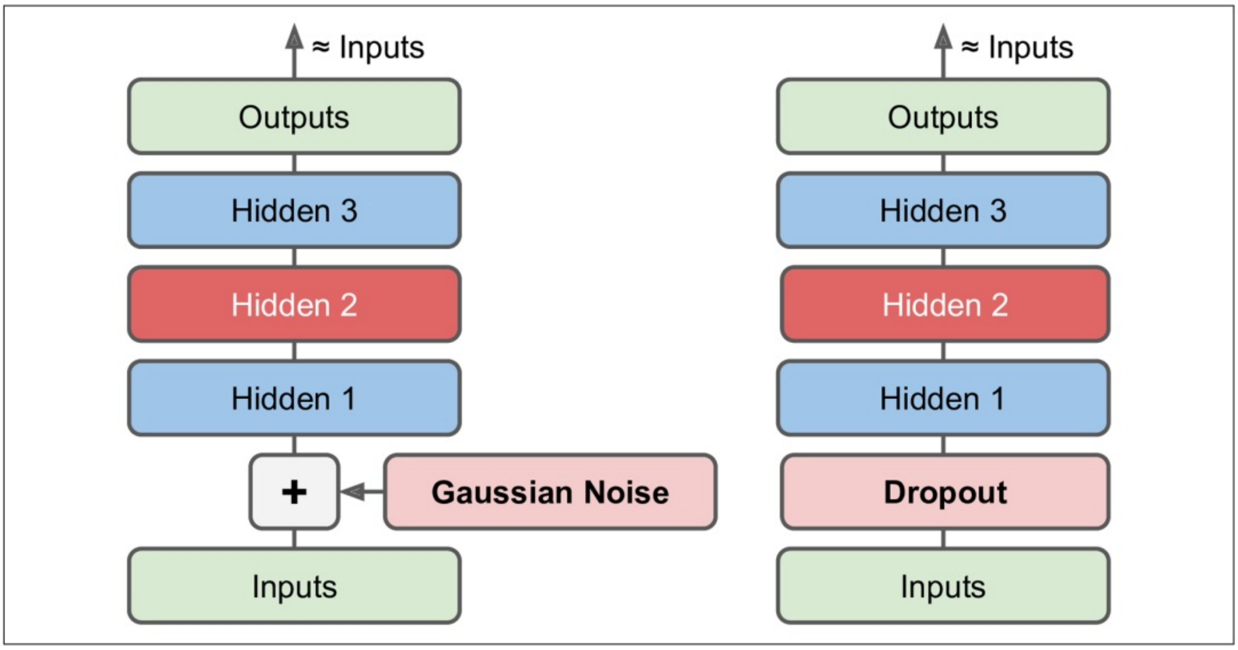


Figure 15-9. Denoising autoencoders, with Gaussian noise (left) or dropout (right)

TensorFlow 实现

在 TensorFlow 中实现去噪自编码器并不难。 我们从高斯噪声开始。 这实际上就像训练一个常规的自编码器一样，除了给输入添加噪声外，重建损耗是根据原始输入计算的：

```
X = tf.placeholder(tf.float32, shape=[None, n_inputs])
X_noisy = X + tf.random_normal(tf.shape(X))
[...]
hidden1 = activation(tf.matmul(X_noisy, weights1) + biases1)
[...]
reconstruction_loss = tf.reduce_mean(tf.square(outputs - X)) # MSE
[...]
```

由于 `X` 的形状只是在构造阶段部分定义的，我们不能预先知道我们必须添加到 `X` 中的噪声的形状。我们不能调用 `X.get_shape()` ，因为这只会返回部分定义的 `X` 的形状（ `[None，n_inputs]` ）和 `random_normal()` 需要一个完全定义的形状，因此会引发异常。 相反，我们调用 `tf.shape(X)` ，它将创建一个操作，该操作将在运行时返回 `X` 的形状，该操作将在此时完全定义。

实施更普遍的 dropout 版本,而且这个版本并不困难：

```
from tensorflow.contrib.layers import dropout

keep_prob = 0.7

is_training = tf.placeholder_with_default(False, shape=(), name='is_training')
X = tf.placeholder(tf.float32, shape=[None, n_inputs])
X_drop = dropout(X, keep_prob, is_training=is_training)
[...]
hidden1 = activation(tf.matmul(X_drop, weights1) + biases1)
[...]
reconstruction_loss = tf.reduce_mean(tf.square(outputs - X)) # MSE
[...]
```

在训练期间，我们必须使用 `feed_dict` 将 `is_training` 设置为 `True` （如第 11 章所述）：

```
sess.run(training_op, feed_dict={X: X_batch, is_training: True})
```

但是，在测试期间，不需要将 `is_training` 设置为 `False` ，因为我们将其设置为对 `placeholder_with_default()` 函数调用的默认值。

稀疏自编码器

通常良好特征提取的另一种约束是稀疏性：通过向损失函数添加适当的项，自编码器被推动以减少编码层中活动神经元的数量。 例如，它可能被推到编码层中平均只有 5% 的显着活跃的神经元。 这迫使自编码器将每个输入表示为少量激活的组合。 因此，编码层中的每个神经元通常都会代表一个有用的特征（如果您每个月只能说几个字，您可能会试着让它们值得一听）。

为了支持稀疏模型，我们必须首先在每次训练迭代中测量编码层的实际稀疏度。 我们通过计算整个训练批次中编码层中每个神经元的平均激活来实现。 批量大小不能太小，否则平均数不准确。

一旦我们对每个神经元进行平均激活，我们希望通过向损失函数添加稀疏损失来惩罚太活跃的神经元。 例如，如果我们测量一个神经元的平均激活值为 0.3，但目标稀疏度为 0.1，那么它必须受到惩罚才能激活更少。 一种方法可以简单地将平方误差 `(0.3-0.1)^2` 添加到损失函数中，但实际上更好的方法是使用 Kullback-Leibler 散度（在第 4 章中简要讨论），其具有比平均值更强的梯度 平方误差，如图 15-10 所示。

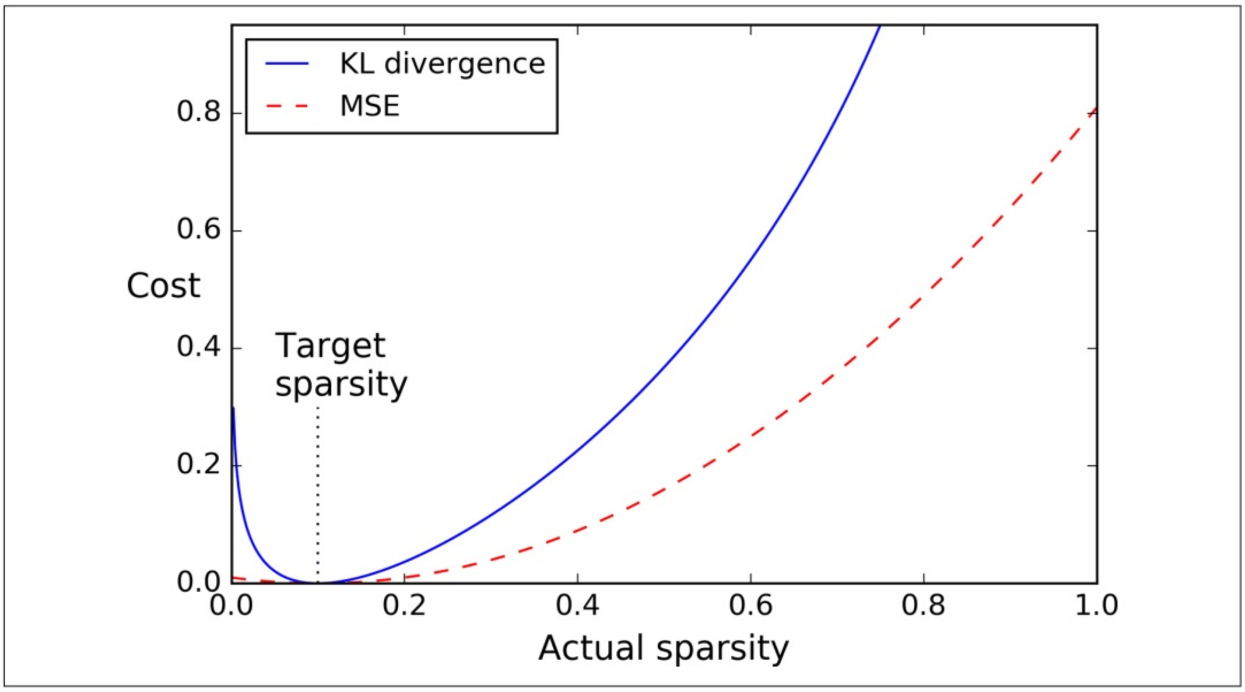


Figure 15-10. Sparsity loss

给定两个离散的概率分布 `P` 和 `Q` ，这些分布之间的 KL 散度，记为 `Dkl(P || Q)` ，可以使用公式 15-1 计算。

Equation 15-1. Kullback–Leibler divergence

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

在我们的例子中，我们想要测量编码层中的神经元将激活的目标概率 `p` 与实际概率 `q` （即，训练批次上的平均激活）之间的差异。 所以KL散度简化为公式 15-2。

Equation 15-2. KL divergence between the target sparsity p and the actual sparsity q

$$D_{\mathrm{KL}}(p \parallel q) = p \log \frac{p}{q} + (1-p) \log \frac{1-p}{1-q}$$

一旦我们已经计算了编码层中每个神经元的稀疏损失，我们就总结这些损失，并将结果添加到损失函数中。 为了控制稀疏损失和重构损失的相对重要性，我们可以用稀疏权重超参数乘以稀疏损失。 如果这个权重太高，模型会紧贴目标稀疏度，但它可能无法正确重建输入，导致模型无用。 相反，如果它太低，模型将大多忽略稀疏目标，它不会学习任何有趣的功能。

TensorFlow 实现

我们现在拥有了使用 TensorFlow 实现稀疏自编码器所需的全部功能：

```
def kl_divergence(p, q):
    return p * tf.log(p / q) + (1 - p) * tf.log((1 - p) / (1 - q))

learning_rate = 0.01
sparsity_target = 0.1
sparsity_weight = 0.2

[...] # Build a normal autoencoder (in this example the coding layer is hidden1)

optimizer = tf.train.AdamOptimizer(learning_rate)

hidden1_mean = tf.reduce_mean(hidden1, axis=0) # batch mean
sparsity_loss = tf.reduce_sum(kl_divergence(sparsity_target, hidden1_mean))
reconstruction_loss = tf.reduce_mean(tf.square(outputs - X)) # MSE
loss = reconstruction_loss + sparsity_weight * sparsity_loss
training_op = optimizer.minimize(loss)
```

一个重要的细节是编码层的激活必须介于 0 和 1 之间（但不等于 0 或 1），否则 KL 散度将返回 `NaN`（非数字）。一个简单的解决方案是对编码层使用逻辑激活功能：

```
hidden1 = tf.nn.sigmoid(tf.matmul(X, weights1) + biases1)
```

一个简单的技巧可以加速收敛：不是使用 MSE，我们可以选择一个具有较大梯度的重建损失。交叉熵通常是一个不错的选择。要使用它，我们必须对输入进行规范化处理，使它们的取值范围为 0 到 1，并在输出层中使用逻辑激活函数，以便输出也取值为 0 到 1。TensorFlow 的 `sigmoid_cross_entropy_with_logits()` 函数负责 有效地将 logistic（sigmoid）激活函数应用于输出并计算交叉熵：

```
[...]
logits = tf.matmul(hidden1, weights2) + biases2)
    outputs = tf.nn.sigmoid(logits)

reconstruction_loss = tf.reduce_sum(
    tf.nn.sigmoid_cross_entropy_with_logits(labels=X, logits=logits))
```

请注意，训练期间不需要输出操作（我们仅在我们想要查看重建时才使用它）。

变分自编码器（VAE）

Diederik Kingma 和 Max Welling 于 2014 年推出了另一类重要的自编码器，并迅速成为最受欢迎的自编码器类型之一：变分自编码器。

它们与我们迄今为止讨论的所有自编码器完全不同，特别是：

- 它们是概率自编码器，意味着即使在训练之后，它们的输出部分也是偶然确定的（相对于仅在训练过程中使用随机性的自编码器的去噪）。
- 最重要的是，它们是生成自编码器，这意味着它们可以生成看起来像从训练集中采样的新实例。

这两个属性使它们与 RBM 非常相似（见附录 E），但它们更容易训练，并且取样过程更快（在 RBM 之前，您需要等待网络稳定在“热平衡”之后才能进行取样一个新的实例）

我们来看看他们是如何工作的。 图 15-11（左）显示了一个变分自编码器。 当然，您可以认识到所有自编码器的基本结构，编码器后跟解码器（在本例中，它们都有两个隐藏层），但有一个转折点：不是直接为给定的输入生成编码 ，编码器产生平均编码 μ 和标准差 σ 。 然后从平均值 μ 和标准差 σ 的高斯分布随机采样实际编码。 之后，解码器正常解码采样的编码。 该图的右侧部分显示了一个训练实例通过此自编码器。 首先，编码器产生 μ 和 σ ，随后对编码进行随机采样（注意它不是完全位于 μ 处），最后对编码进行解码，最终的输出与训练实例类似。

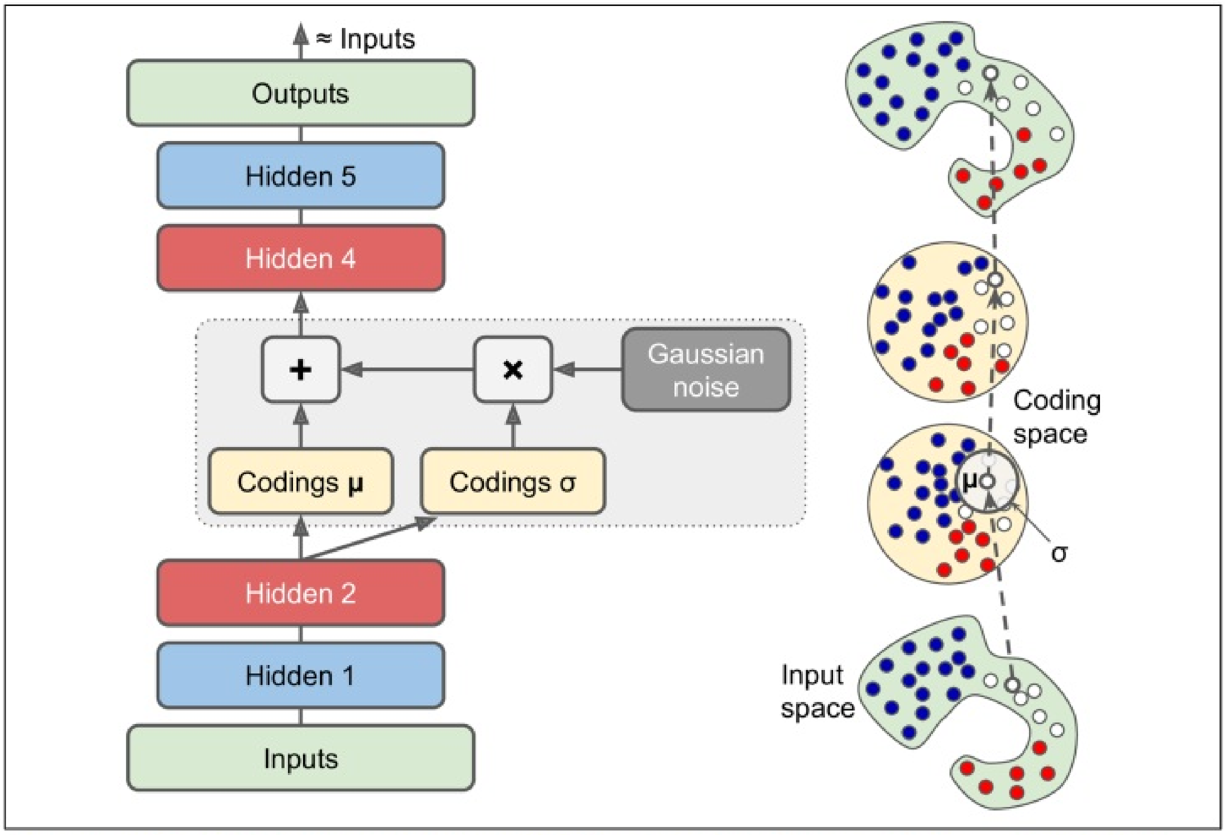


Figure 15-11. Variational autoencoder (left), and an instance going through it (right)

从图中可以看出，尽管输入可能具有非常复杂的分布，但变分自编码器倾向于产生编码，看起来好像它们是从简单的高斯分布采样的：在训练期间，损失函数（将在下面讨论）推动 编码在编码空间（也称为潜在空间）内逐渐迁移以占据看起来像高斯点集成的云的大致（超）球形区域。 一个重要的结果是，在训练了一个变分自编码器之后，你可以很容易地生成一个新的实例：只需从高斯分布中抽取一个随机编码，对它进行解码就可以了！

那么让我们看看损失函数。 它由两部分组成。 首先是通常的重建损失，推动自编码器重现其输入（我们可以使用交叉熵来解决这个问题，如前所述）。 第二种是潜在的损失，推动自编码器使编码看起来像是从简单的高斯分布中采样，为此我们使用目标分布（高斯分布）与编

码实际分布之间的 KL 散度。 数学比以前复杂一点，特别是因为高斯噪声，它限制了可以传输到编码层的信息量（从而推动自编码器学习有用的特征）。 幸运的是，这些方程简化为下面的潜在损失代码：

```
eps = 1e-10 # smoothing term to avoid computing log(0) which is NaN
latent_loss = 0.5 * tf.reduce_sum(
    tf.square(hidden3_sigma) + tf.square(hidden3_mean)
    - 1 - tf.log(eps + tf.square(hidden3_sigma)))
```

一种常见的变体是训练编码器输出 `γ= log(σ^2)` 而不是 `σ` 。 只要我们需要 `σ` ，我们就可以计算 `σ= exp(2/γ)` 。 这使得编码器可以更轻松地捕获不同比例的 `σ` ，从而有助于加快收敛速度。 潜在损失结束会变得更简单一些：

```
latent_loss = 0.5 * tf.reduce_sum(
    tf.exp(hidden3_gamma) + tf.square(hidden3_mean) - 1 - hidden3_gamma)
```

以下代码使用 `log(σ^2)` 变体构建图 15-11（左）所示的变分自编码器:

```
n_inputs = 28 * 28 # for MNIST
n_hidden1 = 500
n_hidden2 = 500
n_hidden3 = 20 # codings
n_hidden4 = n_hidden2
n_hidden5 = n_hidden1
n_outputs = n_inputs

learning_rate = 0.001

with tf.contrib.framework.arg_scope(
        [fully_connected],
        activation_fn=tf.nn.elu,
        weights_initializer=tf.contrib.layers.variance_scaling_initializer()):
    X = tf.placeholder(tf.float32, [None, n_inputs])
    hidden1 = fully_connected(X, n_hidden1)
    hidden2 = fully_connected(hidden1, n_hidden2)
    hidden3_mean = fully_connected(hidden2, n_hidden3, activation_fn=None)
    hidden3_gamma = fully_connected(hidden2, n_hidden3, activation_fn=None)
    hidden3_sigma = tf.exp(0.5 * hidden3_gamma)
    noise = tf.random_normal(tf.shape(hidden3_sigma), dtype=tf.float32)
    hidden3 = hidden3_mean + hidden3_sigma * noise
    hidden4 = fully_connected(hidden3, n_hidden4)
    hidden5 = fully_connected(hidden4, n_hidden5)
    logits = fully_connected(hidden5, n_outputs, activation_fn=None)
    outputs = tf.sigmoid(logits)
reconstruction_loss = tf.reduce_sum(
    tf.nn.sigmoid_cross_entropy_with_logits(labels=X, logits=logits))
latent_loss = 0.5 * tf.reduce_sum(
    tf.exp(hidden3_gamma) + tf.square(hidden3_mean) - 1 - hidden3_gamma)
cost = reconstruction_loss + latent_loss

optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(cost)

init = tf.global_variables_initializer()
```

生成数字

现在让我们使用这个变分自编码器来生成看起来像手写数字的图像。 我们所需要做的就是训练模型，然后从高斯分布中对随机编码进行采样并对它们进行解码。

```
import numpy as np
n_digits = 60
n_epochs = 50
batch_size = 150
with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        n_batches = mnist.train.num_examples // batch_size
        for iteration in range(n_batches):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch})

    codings_rnd = np.random.normal(size=[n_digits, n_hidden3])
    outputs_val = outputs.eval(feed_dict={hidden3: codings_rnd})
```

现在我们可以看到由autoencoder生成的“手写”数字是什么样的（参见图15-12）：

```
for iteration in range(n_digits):
    plt.subplot(n_digits, 10, iteration + 1)
    plot_image(outputs_val[iteration])
```

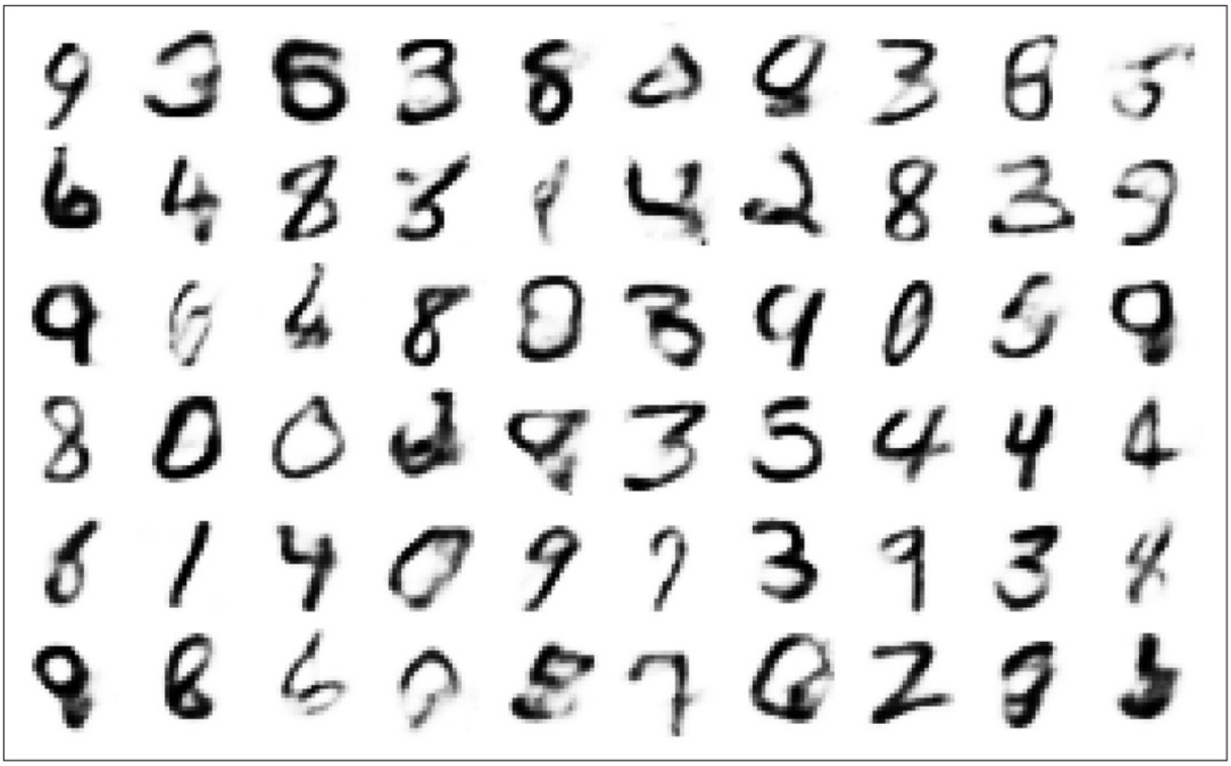


Figure 15-12. Images of handwritten digits generated by the variational autoencoder

其他自编码器

监督式学习在图像识别，语音识别，文本翻译等方面取得的惊人成就在某种程度上掩盖了无监督学习的局面，但它实际上正在蓬勃发展。 自编码器和其他无监督学习算法的新体系结构定期发明，以至于我们无法在本书中全面介绍它们。 以下是您可能想要查看的几种类型的自编码器的简要说明（绝非详尽无遗）：

压缩自编码器（CAE）

自编码器在训练过程中受到约束，因此与输入有关的编码的导数很小。 换句话说，两个类似的输入必须具有相似的编码。

栈式卷积自编码器（SCAE）

学习通过重构通过卷积层处理的图像来提取视觉特征的自编码器。

生成随机网络（GSN）

消除自编码器的泛化，增加了生成数据的能力。

赢家通吃（WTA）的自编码

在训练期间，在计算编码层中所有神经元的激活之后，只保留训练批次上每个神经元的前 k% 激活，其余部分设为零。 自然这导致稀疏的编码。 而且，可以使用类似的 WTA 方法来产生稀疏卷积自编码器。

对抗自编码器（AAE）

一个网络被训练来重现它的输入，同时另一个网络被训练去找到第一个网络不能正确重建的输入。 这推动了第一个自编码器学习健壮的编码。

第16章 强化学习

强化学习（RL）如今是机器学习的一大令人激动的领域，当然之前也是。自从 1950 年被发明出来后，它在这些年产生了一些有趣的应用，尤其是在游戏（例如 TD-Gammon，一个西洋双陆棋程序）和及其控制领域，但是从未弄出什么大新闻。直到 2013 年一个革命性的发展：来自英国的研究者发起了一项 Deepmind 项目，这个项目可以学习去玩任何从头开始的 Atari 游戏，甚至多数比人类玩的还要好，它仅适用像素作为输入并且没有游戏规则的任何先验知识。这是一系列令人惊叹的壮举，在 2016 年 3 月以他们的系统阿尔法狗战胜了世界围棋冠军李世石。没有一个程序能接近这个游戏的主宰，更不用说世界冠军了。今天，RL 的整个领域正在沸腾着新的想法，其都具有广泛的应用范围。DeepMind 在 2014 被谷歌以超过 5 亿美元收购。

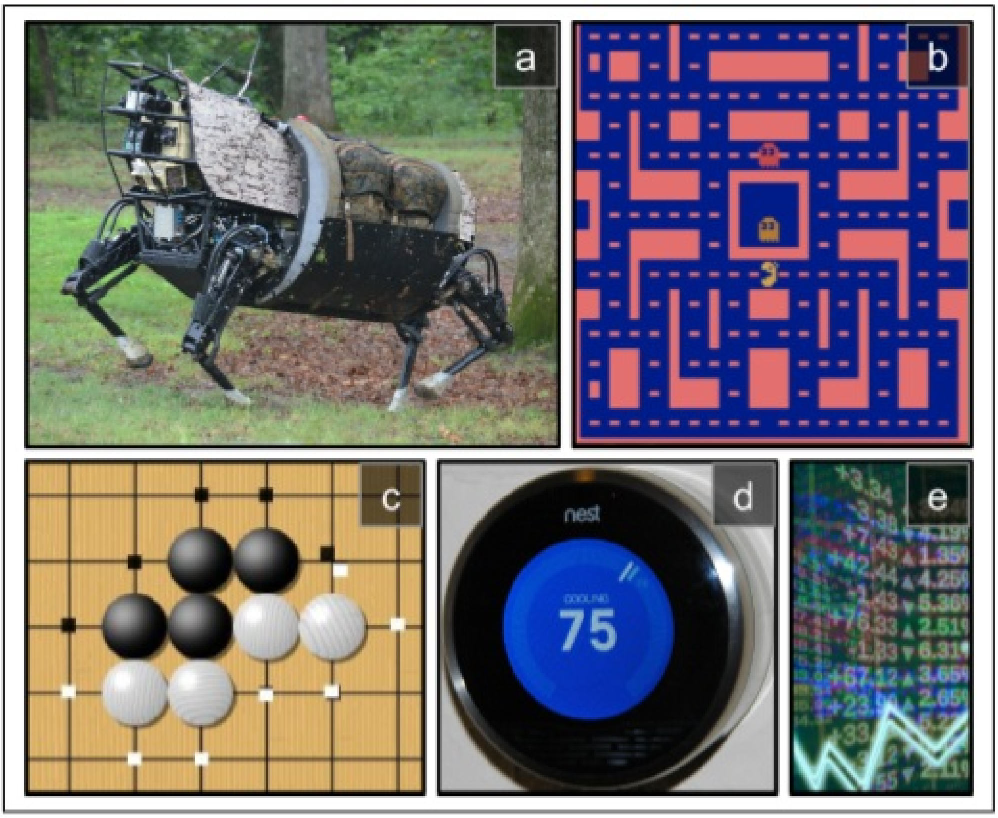
那么他们是怎么做到的呢？事后看来，原理似乎相当简单：他们将深度学习运用到强化学习领域，结果却超越了他们最疯狂的设想。在本章中，我们将首先解释强化学习是什么，以及它擅长于什么，然后我们将介绍两个在深度强化学习领域最重要的技术：策略梯度和深度 Q 网络（DQN），包括讨论马尔可夫决策过程（MDP）。我们将使用这些技术来训练一个模型来平衡移动车上的杆子，另一个玩 Atari 游戏。同样的技术可以用于各种各样的任务，从步行机器人到自动驾驶汽车。

学习优化奖励

在强化学习中，智能体在环境中观察并且做出决策，随后它会得到奖励。它的目标是去学习如何行动能最大化期望的奖励。如果你不在意去拟人化的话，你可以认为正奖励是愉快，负奖励是痛苦（这样的话奖励一词就有点误导了）。简单来说，智能体在环境中行动，并且在实验和错误中去学习最大化它的愉快，最小化它的痛苦。

这是一个相当广泛的设置，可以适用于各种各样的任务。以下是几个例子（详见图 16-1）：

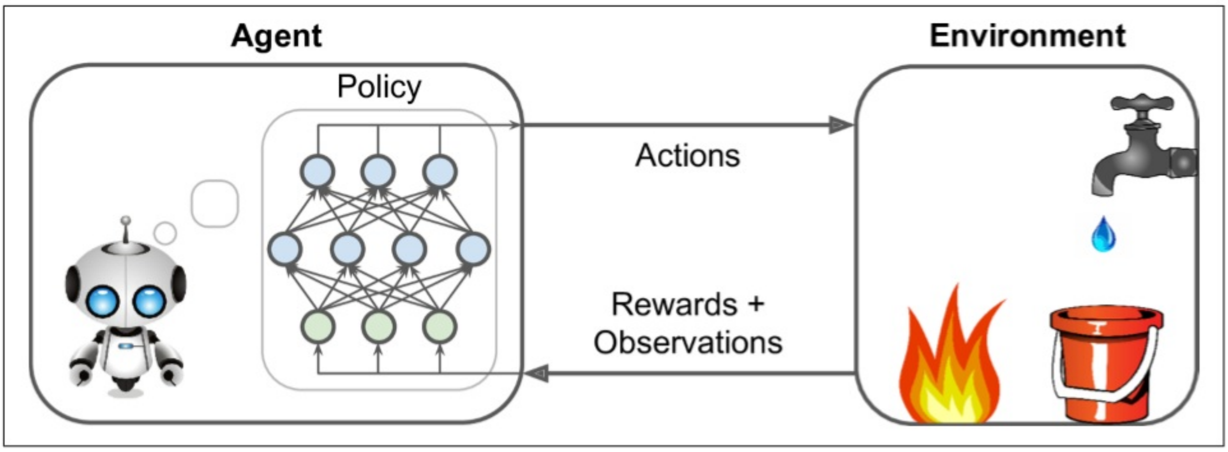
1. 智能体可以是控制一个机械狗的程序。在此例中，环境就是真实的世界，智能体通过许多的传感器例如摄像机或者传感器来观察，它可以通过给电机驱动信号来行动。它可以被编程设置为如果到达了目的地就得到正奖励，如果浪费时间，或者走错方向，或摔倒了就得到负奖励。
2. 智能体可以是控制 MS.Pac-Man 的程序。在此例中，环境是 Atari 游戏的仿真，行为是 9 个操纵杆位（上下左右中间等等），观察是屏幕，回报就是游戏点数。
3. 相似地，智能体也可以是棋盘游戏的程序例如：围棋。
4. 智能体也可以不用去控制一个实体（或虚拟的）去移动。例如它可以是一个智能程序，当它调整到目标温度以节能时会得到正奖励，当人们需要自己去调节温度时它会得到负奖励，所以智能体必须学会预见人们的需要。
5. 智能体也可以去观测股票市场价格以实时决定买卖。奖励的依据显然为挣钱或者赔钱。



其实没有正奖励也是可以的，例如智能体在迷宫内移动，它每分每秒都得到一个负奖励，所以它要尽可能快的找到出口！还有很多适合强化学习的领域，例如自动驾驶汽车，在网页上放广告，或者控制一个图像分类系统让它明白它应该关注于什么。

策略搜索

被智能体使用去改变它行为的算法叫做策略。例如，策略可以是一个把观测当输入，行为当做输出的神经网络。

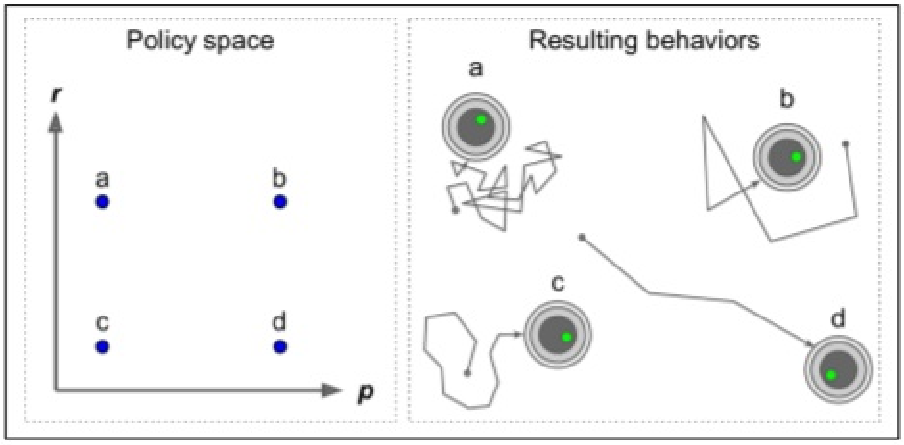


这个策略可以是你能想到的任何算法，它甚至可以不被确定。举个例子，例如，考虑一个真空吸尘器，它的奖励是在 30 分钟内捡起的灰尘数量。它的策略可以是每秒以概率 `p` 向前移动，或者以概率 `1-p` 随机地向左或向右旋转。旋转角度将是 `-r` 和 `+r` 之间的随机角度，因为

该策略涉及一些随机性，所以称为随机策略。机器人将有一个不稳定的轨迹，它保证它最终会到达任何可以到达的地方，并捡起所有的灰尘。问题是：30分钟后它会捡起多少灰尘？

你怎么训练这样的机器人？你可以调整两个策略参数：概率 `P` 和角度范围 `R` 。一个想法是这些参数尝试许多不同的值，并选择执行最佳的组合（见图 16-3）。这是一个策略搜索的例子，在这种情况下使用野蛮的方法。然而，当策略空间太大（通常情况下），以这样的方式找到一组好的参数就像是大海捞针。

另一种搜寻策略空间的方法是遗传算法。例如你可以随机创造一个包含 100 个策略的第一代基因，随后杀死 80 个坏策略，随后让 20 个幸存策略繁衍 4 代。一个后代只是它父辈基因的复制品加上一些随机变异。幸存的策略加上他们的后代共同构成了第二代。你可以继续以这种方式迭代代，直到找到一个好的策略。



另一种方法是使用优化技术，通过评估关于策略参数的奖励的梯度，然后通过跟随梯度向更高的奖励（梯度上升）调整这些参数。这种方法被称为策略梯度（PG），我们将在本章后面详细讨论。例如，回到真空吸尘器机器人，你可以稍微增加P并评估这是否增加了机器人在 30 分钟内拾起的灰尘的量；如果它相对应增加 `P` ，或者减少 `P` 。我们将使用 Tensorflow 来实现 PG 算法，但是在这之前我们需要为智能体创造一个生存的环境，所以现在是介绍 OpenAI 的时候了。

OpenAI 的介绍

强化学习的一个挑战是，为了训练智能体，首先需要有一个工作环境。如果你想设计一个可以学习 Atari 游戏的程序，你需要一个 Atari 游戏模拟器。如果你想设计一个步行机器人，那么环境就是真实的世界，你可以直接在这个环境中训练你的机器人，但是这有其局限性：如果机器人从悬崖上掉下来，你不能仅仅点击“撤消”。你也不能加快时间；增加更多的计算能力不会让机器人移动得更快。一般来说，同时训练 1000 个机器人是非常昂贵的。简而言之，训练在现实世界中是困难和缓慢的，所以你通常需要一个模拟环境，至少需要引导训练。

OpenAI gym 是一个工具包，它提供各种各样的模拟环境（Atari 游戏，棋盘游戏，2D 和 3D 物理模拟等等），所以你可以训练，比较，或开发新的 RL 算法。

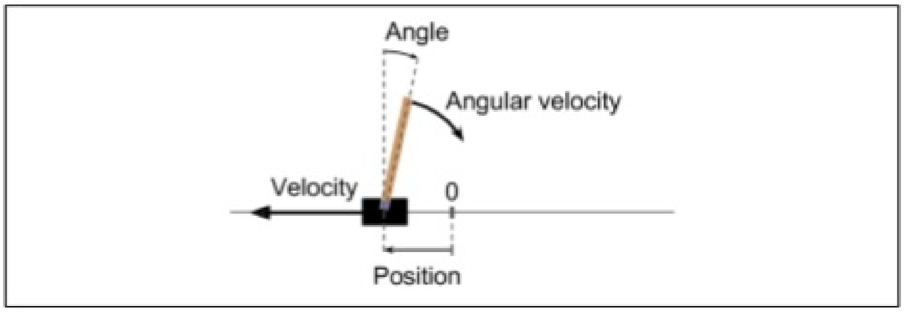
让我们安装 OpenAI gym。可通过 `pip` 安装：

```
$ pip install --upgrade gym
```

接下来打开 Python shell 或 Jupyter 笔记本创建您的第一个环境：

```
>>> import gym
>>> env = gym.make("CartPole-v0")
[2016-10-14 16:03:23,199] Making new env: MsPacman-v0
>>> obs = env.reset()
>>> obs
array([-0.03799846,-0.03288115,0.02337094,0.00720711])
>>> env.render()
```

使用 `make()` 函数创建一个环境，在此例中是 CartPole 环境。这是一个 2D 模拟，其中推车可以被左右加速，以平衡放置在它上面的平衡杆（见图 16-4）。在创建环境之后，我们需要使用 `reset()` 初始化。这会返回第一个观察结果。观察取决于环境的类型。对于 CartPole 环境，每个观测是包含四个浮点的 1D Numpy 向量：这些浮点数代表推车的水平位置（0 为中心）、其速度、杆的角度（0 维垂直）及其角速度。最后，`render()` 方法显示如图 16-4 所示的环境。



如果你想让 `render()` 让图像以一个 NUMPY 数组格式返回，可以将 `mode` 参数设置为 `rgb_array`（注意其他环境可能支持不同的模式）：

```
>>> img = env.render(mode="rgb_array")
>>> img.shape  # height, width, channels (3=RGB)
(400, 600, 3)
```

不幸的是，即使将 `mode` 参数设置为 `rgb_array`，CartPole（和其他一些环境）还是会将将图像呈现到屏幕上。避免这种情况的唯一方式是使用一个 fake X 服务器，如 XVFB 或 XDimMy。例如，可以使用以下命令安装 XVFB 和启动 Python：`xvfb-run -s "screen 0 1400x900x24" python`。或者使用 `xvfbwrapper` 包。

让我们来询问环境什么动作是可能的：

```
>>> env.action_space
Discrete(2)
```

`Discrete(2)` 表示可能的动作是整数 0 和 1，表示向左（0）或右（1）的加速。其他环境可能有更多的动作，或者其他类型的动作（例如，连续的）。因为杆子向右倾斜，让我们向右加速推车：

```
>>> action = 1  # accelerate right
>>> obs, reward, done, info = env.step(action)
>>> obs
array([-0.03865608,  0.16189797,  0.02351508, -0.27801135])
>>> reward
1.0
>>> done
False
>>> info
{}
```

`step()` 表示执行给定的动作并返回四个值：

`obs`：

这是新的观测，小车现在正在向右走（ `obs[1]>0` ）。平衡杆仍然向右倾斜（ `obs[2]>0` ），但是他的角速度现在为负（ `obs[3]<0` ），所以它在下一步后可能会向左倾斜。

`reward` ：

在这个环境中，无论你做什么，每一步都会得到 1.0 奖励，所以游戏的目标就是尽可能长的运行。

`done` ：

当游戏结束时这个值会为 `True` 。当平衡单倾斜太多时会发生这种情况。之后，必须重新设置环境才能重新使用。

`info` ：

该字典可以在其他环境中提供额外的调试信息。这些数据不应该用于训练（这是作弊）。

让我们编码一个简单的策略，当杆向左倾斜时加速左边，当杆向右倾斜时加速。我们使用这个策略来获得超过 500 步的平均回报：

```
def basic_policy(obs):
    angle = obs[2]
    return 0 if angle < 0 else 1
totals = []
for episode in range(500):
    episode_rewards = 0
    obs = env.reset()
    for step in range(1000): # 最多1000 步，我们不想让它永远运行下去
        action = basic_policy(obs)
        obs, reward, done, info = env.step(action)
        episode_rewards += reward
        if done:
            break
    totals.append(episode_rewards)
```

这个代码希望能自我解释。让我们看看结果：

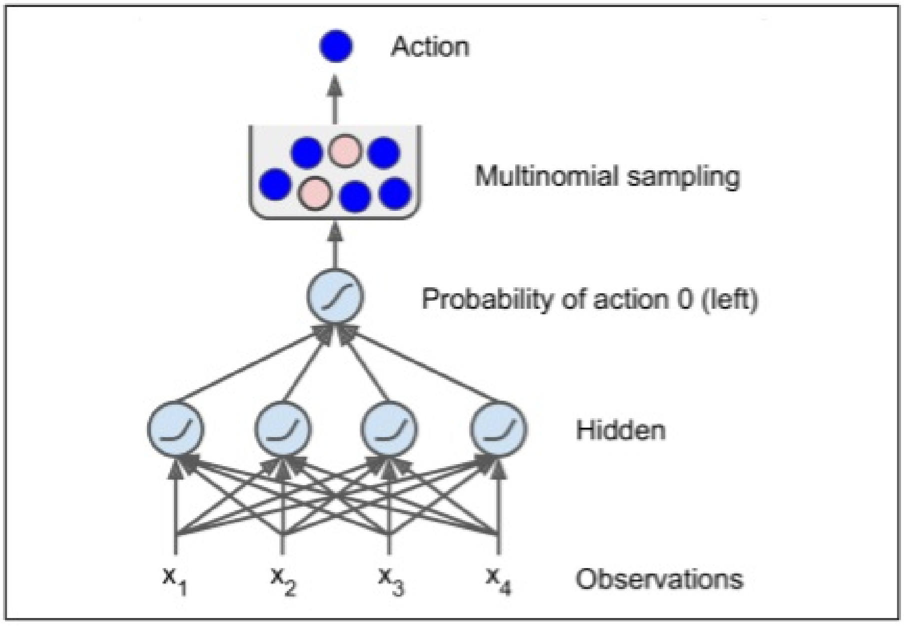
```
>>> import numpy as np
>>> np.mean(totals), np.std(totals), np.min(totals), np.max(totals) (42.125999999999998
, 9.1237121830974033, 24.0, 68.0)
```

即使有 500 次尝试，这一策略从未超过 68 个连续的步骤保持平衡杆直立。这不太好。如果你看一下 Juyter Notebook 中的模拟，你会发现，推车越来越强烈地左右摆动，直到平衡杆倾斜太多。让我们看看神经网络是否能提出更好的策略。

神经网络策略

让我们创建一个神经网络策略。就像之前我们编码的策略一样，这个神经网络将把观察作为输入，输出要执行的动作。更确切地说，它将估计每个动作的概率，然后我们将根据估计的概率随机地选择一个动作（见图 16-5）。在 CartPole 环境中，只有两种可能的动作（左或右），所以我们只需要一个输出神经元。它将输出动作 0（左）的概率 `p`，动作 1（右）的概率显然将是 `1 - p`。

例如，如果它输出 0.7，那么我们将以 70% 的概率选择动作 0，以 30% 的概率选择动作 1。



你可能奇怪为什么我们根据神经网络给出的概率来选择随机的动作，而不是选择最高分数的动作。这种方法使这个点在探索新的行为和利用已知的工作的行动之间找到正确的平衡。举个例子：假设你第一次去餐馆，所有的菜看起来同样吸引人，所以你随机挑选一个。如果菜好吃，你可以增加下一次点它的概率，但是你不应该把这个概率提高到 100%，否则你将永远不会尝试其他菜肴，其中一些甚至比你尝试的更好。

还要注意，在这个特定的环境中，过去的动作和观察可以被安全地忽略，因为每个观察都包含环境的完整状态。如果有一些隐藏状态，那么你也需要考虑过去的行为和观察。例如，如果环境仅仅揭示了推车的位置，而不是它的速度，那么你不仅要考虑当前的观测，还要考虑先前的观测，以便估计当前的速度。另一个例子是当观测是有噪声的的，在这种情况下，通常你想用过去的观察来估计最可能的当前状态。因此，CartPole 问题是简单的；观测是无噪声的，而且它们包含环境的全状态。

```
import tensorflow as tf
from tensorflow.contrib.layers import fully_connected
# 1. 声明神经网络结构
n_inputs = 4  # == env.observation_space.shape[0]
n_hidden = 4  # 这只是个简单的测试，不需要过多的隐藏层
n_outputs = 1 # 只输出向左加速的概率
initializer = tf.contrib.layers.variance_scaling_initializer()
# 2. 建立神经网络
X = tf.placeholder(tf.float32, shape=[None, n_inputs]) hidden = fully_connected(X, n_h
idden, activation_fn=tf.nn.elu,weights_initializer=initializer)
logits = fully_connected(hidden, n_outputs, activation_fn=None,weights_initializer=ini
tializer)
outputs = tf.nn.sigmoid(logits)
# 3. 在概率基础上随机选择动作
p_left_and_right = tf.concat(axis=1, values=[outputs, 1 - outputs])
action = tf.multinomial(tf.log(p_left_and_right), num_samples=1)
init = tf.global_variables_initializer()
```

让我们通读代码：

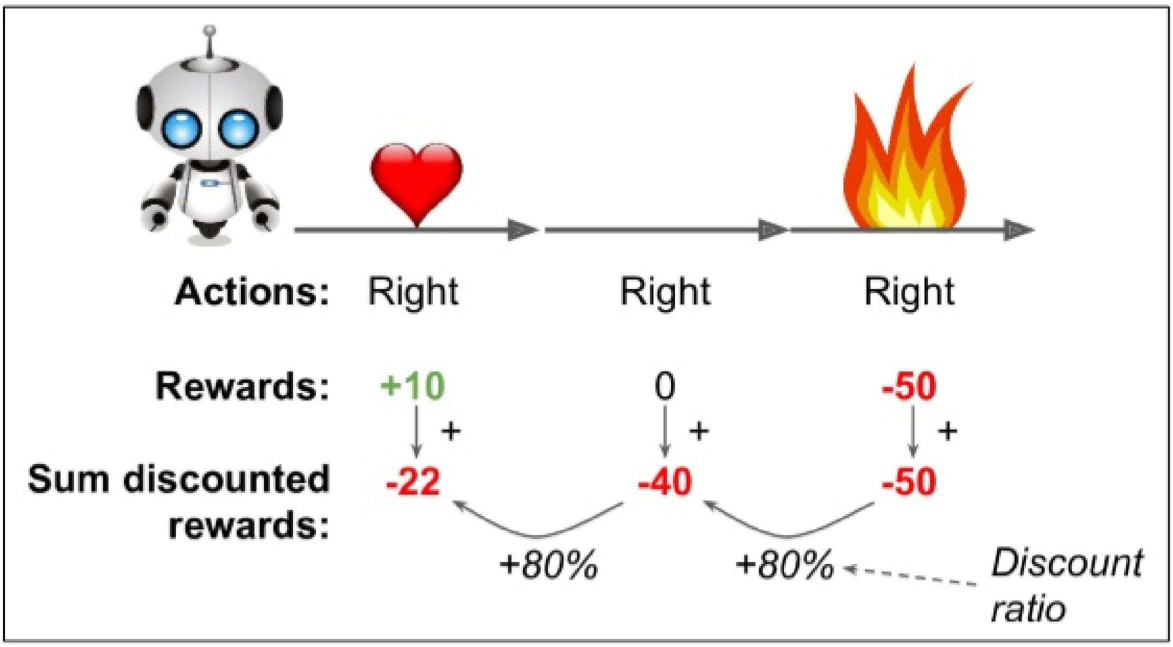
1. 在导入之后，我们定义了神经网络体系结构。输入的数量是观测空间的大小（在 CartPole 的情况下是 4 个），我们只有 4 个隐藏单元，并且不需要更多，并且我们只有 1 个输出概率（向左的概率）。
2. 接下来我们构建了神经网络。在这个例子中，它是一个 vanilla 多层感知器，只有一个输出。注意，输出层使用 Logistic（Sigmoid）激活函数，以便输出从 0 到 1 的概率。如果有两个以上的可能动作，每个动作都会有一个输出神经元，你将使用 Softmax 激活函数。
3. 最后，我们调用 `multinomial()` 函数来选择一个随机动作。该函数独立地采样一个（或多个）整数，给定每个整数的对数概率。例如，如果通过设置 `num_samples=5`，令数组为 `[np.log(0.5), np.log(0.2), np.log(0.3)]` 来调用它，那么它将输出五个整数，每个整数都有 50% 的概率是 0，20% 为 1，30% 为 2。在我们的情况下，我们只需要一个整数来表示要采取的行动。由于输出张量仅包含向左的概率，所以我们必须首先将 1 个输出连接到它，以得到包含左和右动作的概率的张量。请注意，如果有两个以上的可能动作，神经网络将不得不输出每个动作的概率，这样你就不需要连接步骤了。

好了，现在我们有一个可以观察和输出动作的神经网络了，那我们怎么训练它呢？

评价行为：信用分配问题

如果我们知道每一步的最佳动作，我们可以像通常一样训练神经网络，通过最小化估计概率和目标概率之间的交叉熵。这只是通常的监督学习。然而，在强化学习中，智能体获得的指导的唯一途径是通过奖励，奖励通常是稀疏的和延迟的。例如，如果智能体在 100 个步骤内设法平衡杆，它怎么知道它采取的 100 个行动中的哪一个是好的，哪些是坏的？它所知道的是，在最后一次行动之后，杆子坠落了，但最后一次行动肯定不是完全负责的。这被称为信用分配问题：当智能体得到奖励时，很难知道哪些行为应该被信任（或责备）。想想一只狗在行为良好后几小时就会得到奖励，它会明白它得到了什么回报吗？

为了解决这个问题，一个通常的策略是基于这个动作后得分的总和来评估这个个动作，通常在每个步骤中应用衰减率 `r` 。例如（见图 16-6），如果一个智能点决定连续三次向右，在第一步之后得到 +10 奖励，第二步后得到 0，最后在第三步之后得到 -50，然后假设我们使用衰减率 `r=0.8` ，那么第一个动作将得到 `10 +r×0 + r2×(-50)=-22` 的分述。如果衰减率接近 0，那么与即时奖励相比，未来的奖励不会有多大意义。相反，如果衰减率接近 1，那么对未来的奖励几乎等于即时回报。典型的衰减率通常为是 0.95 或 0.99。如果衰减率为 0.95，那么未来 13 步的奖励大约是即时奖励的一半（ `0.9513×0.5` ），而当衰减率为 0.99，未来 69 步的奖励是即时奖励的一半。在 CartPole 环境下，行为具有相当短期的影响，因此选择 0.95 的折扣率是合理的。



当然，一个好的动作可能会伴随着一些坏的动作，这些动作会导致平衡杆迅速下降，从而导致一个好的动作得到一个低分数（类似的，一个好演员有时会在一部烂片中扮演主角）。然而，如果我们花足够多的时间来训练游戏，平均下来好的行为会得到比坏的更好的分数。因此，为了获得相当可靠的动作分数，我们必须运行很多次并将所有动作分数归一化（通过减去平均值并除以标准偏差）。之后，我们可以合理地假设消极得分的行为是坏的，而积极得分的行为是好的。现在我们有一个方法来评估每一个动作，我们已经准备好使用策略梯度来训练我们的第一个智能体。让我们看看如何。

策略梯度

正如前面所讨论的，PG 算法通过遵循更高回报的梯度来优化策略参数。一种流行的 PG 算法，称为增强算法，在 1929 由 Ronald Williams 提出。这是一个常见的变体：

1. 首先，让神经网络策略玩几次游戏，并在每一步计算梯度，这使得智能点更可能选择行为，但不应用这些梯度。

2. 运行几次后，计算每个动作的得分（使用前面段落中描述的方法）。

3. 如果一个动作的分数是正的，这意味着动作是好的，可应用较早计算的梯度，以便将来有更大的的概率选择这个动作。但是，如果分数是负的，这意味着动作是坏的，要应用负梯度来使得这个动作在将来采取的可能性更低。我们的方法就是简单地将每个梯度向量乘以相应的动作得分。

4. 最后，计算所有得到的梯度向量的平均值，并使用它来执行梯度下降步骤。

让我们使用 TensorFlow 实现这个算法。我们将训练我们早先建立的神经网络策略，让它学会平衡车上的平衡杆。让我们从完成之前编码的构造阶段开始，添加目标概率、代价函数和训练操作。因为我们的意愿是选择的动作是最好的动作，如果选择的动作是动作 0（左），则目标概率必须为 1，如果选择动作 1（右）则目标概率为 0：

```
y = 1. - tf.to_float(action)
```

现在我们有一个目标概率，我们可以定义损失函数（交叉熵）并计算梯度：

```
learning_rate = 0.01
cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(                    labels=y,
logits=logits)
optimizer = tf.train.AdamOptimizer(learning_rate)
grads_and_vars = optimizer.compute_gradients(cross_entropy)
```

注意，我们正在调用优化器的 `compute_gradients()` 方法，而不是 `minimize()` 方法。这是因为我们想要在使用它们之前调整梯度。 `compute_gradients()` 方法返回梯度向量/变量对的列表（每个可训练变量一对）。让我们把所有的梯度放在一个列表中，以便方便地获得它们的值：

```
gradients = [grad for grad, variable in grads_and_vars]
```

好，现在是棘手的部分。在执行阶段，算法将运行策略，并在每个步骤中评估这些梯度张量并存储它们的值。在多次运行之后，它如先前所解释的调整这些梯度（即，通过动作分数乘以它们并使它们归一化），并计算调整后的梯度的平均值。接下来，需要将结果梯度反馈到优化器，以便它可以执行优化步骤。这意味着对于每一个梯度向量我们需要一个占位符。此

外，我们必须创建操作去应用更新的梯度。为此，我们将调用优化器的 `apply_gradients()` 函数，该函数采用梯度向量/变量对的列表。我们不给它原始的梯度向量，而是给它一个包含更新梯度的列表（即，通过占位符递送的梯度）：

```
gradient_placeholders = []
grads_and_vars_feed = []
for grad, variable in grads_and_vars:
    gradient_placeholder = tf.placeholder(tf.float32, shape=grad.get_shape())
    gradient_placeholders.append(gradient_placeholder)
    grads_and_vars_feed.append((gradient_placeholder, variable))
training_op = optimizer.apply_gradients(grads_and_vars_feed)
```

让我们后退一步，看看整个运行过程：

```
n_inputs = 4
n_hidden = 4
n_outputs = 1
initializer = tf.contrib.layers.variance_scaling_initializer()

learning_rate = 0.01
X = tf.placeholder(tf.float32, shape=[None, n_inputs])
hidden = fully_connected(X, n_hidden, activation_fn=tf.nn.elu,weights_initializer=init
ializer)
logits = fully_connected(hidden, n_outputs, activation_fn=None,                  wei
ghts_initializer=initializer)
outputs = tf.nn.sigmoid(logits)
p_left_and_right = tf.concat(axis=1, values=[outputs, 1 - outputs])
action = tf.multinomial(tf.log(p_left_and_right), num_samples=1)

y = 1. - tf.to_float(action)
cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(labels=y, logits=logits)
optimizer = tf.train.AdamOptimizer(learning_rate)
grads_and_vars = optimizer.compute_gradients(cross_entropy)
gradients = [grad for grad, variable in grads_and_vars]
gradient_placeholders = []
grads_and_vars_feed = []
for grad, variable in grads_and_vars:
gradient_placeholder = tf.placeholder(tf.float32, shape=grad.get_shape())    gradient_
placeholders.append(gradient_placeholder)
grads_and_vars_feed.append((gradient_placeholder, variable))
training_op = optimizer.apply_gradients(grads_and_vars_feed)

init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

到执行阶段了！我们将需要两个函数来计算总折扣奖励，给予原始奖励，以及归一化多次循环的结果：

```
def discount_rewards(rewards, discount_rate):
    discounted_rewards = np.empty(len(rewards))
    cumulative_rewards = 0
    for step in reversed(range(len(rewards))):
        cumulative_rewards = rewards[step] + cumulative_rewards * discount_rate
discounted_rewards[step] = cumulative_rewards
    return discounted_rewards

def discount_and_normalize_rewards(all_rewards, discount_rate):
    all_discounted_rewards = [discount_rewards(rewards) for rewards in all_rewards]
    flat_rewards = np.concatenate(all_discounted_rewards)
    reward_mean = flat_rewards.mean()
    reward_std = flat_rewards.std()
    return [(discounted_rewards - reward_mean)/reward_std  for discounted_rewards in a
ll_discounted_rewards]
```

让我们检查一下运行的如何：

```
>>> discount_rewards([10, 0, -50], discount_rate=0.8)
array([-22., -40., -50.])
>>> discount_and_normalize_rewards([[10, 0, -50], [10, 20]], discount_rate=0.8)
[array([-0.28435071, -0.86597718, -1.18910299]), array([ 1.26665318,  1.0727777 ])]
```

对 `discount_rewards()` 的调用正好返回我们所期望的（见图 16-6）。你也可以验证函数 `iscount_and_normalize_rewards()` 确实返回了两个步骤中每个动作的标准化分数。注意第一步比第二步差很多，所以它的归一化分数都是负的；从第一步开始的所有动作都会被认为是坏的，反之，第二步的所有动作都会被认为是好的。

我们现在有了训练策略所需的一切：

```
n_iterations = 250      # 训练迭代次数
n_max_steps = 1000      # 每一次的最大步长
n_games_per_update = 10 # 每十次训练一次策略
save_iterations = 10    # 每十次迭代保存模型
discount_rate = 0.95
with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        all_rewards = []    #每一次的所有奖励
        all_gradients = []  #每一次的所有梯度
        for game in range(n_games_per_update):
            current_rewards = []   #当前步的所有奖励
            current_gradients = [] #当前步的所有梯度
            obs = env.reset()
            for step in range(n_max_steps):
                action_val, gradients_val = sess.run([action, gradients],
                feed_dict={X: obs.reshape(1, n_inputs)}) # 一个obs
                obs, reward, done, info = env.step(action_val[0][0])                cu
rrent_rewards.append(reward)
                current_gradients.append(gradients_val)
                if done:
                    break
                all_rewards.append(current_rewards)
                all_gradients.append(current_gradients)
        # 此时我们每10次运行一次策略，我们已经准备好使用之前描述的算法去更新策略
        all_rewards = discount_and_normalize_rewards(all_rewards)
        feed_dict = {}
        for var_index, grad_placeholder in enumerate(gradient_placeholders):
  # 将梯度与行为分数相乘，并计算平均值
            mean_gradients = np.mean([reward * all_gradients[game_index][step][var_ind
ex] for game_index, rewards in enumerate(all_rewards)             for step, reward in
enumerate(rewards)],axis=0)
            feed_dict[grad_placeholder] = mean_gradients
            sess.run(training_op, feed_dict=feed_dict)
            if iteration % save_iterations == 0:
                saver.save(sess, "./my_policy_net_pg.ckpt")
```

每一次训练迭代都是通过运行10次的策略开始的（每次最多 1000 步，以避免永远运行）。在每一步，我们也计算梯度，假设选择的行动是最好的。在运行了这 10 次之后，我们使用 `discount_and_normalize_rewards()` 函数计算动作得分；我们遍历每个可训练变量，在所有次数和所有步骤中，通过其相应的动作分数来乘以每个梯度向量；并且我们计算结果的平均值。最后，我们运行训练操作，给它提供平均梯度（对每个可训练变量提供一个）。我们继续每 10 个训练次数保存一次模型。

我们做完了！这段代码将训练神经网络策略，它将成功地学会平衡车上的平衡杆（你可以在 Juyter notebook 上试用）。注意，实际上有两种方法可以让玩家游戏结束：要么平衡可以倾斜太大，要么车完全脱离屏幕。在 250 次训练迭代中，策略学会平衡极点，但在避免脱离屏幕方面还不够好。额外数百次的训练迭代可以解决这一问题。

研究人员试图找到一种即使当智能点最初对环境一无所知时也能很好地工作的算法。然而，除非你正在写论文，否则你应该尽可能多地将先前的知识注入到智能点中，因为它会极大地加速训练。例如，你可以添加与屏幕中心距离和极点角度成正比的负奖励。此外，如果你已经有一个相当好的策略，你可以训练神经网络模仿它，然后使用策略梯度来改进它。

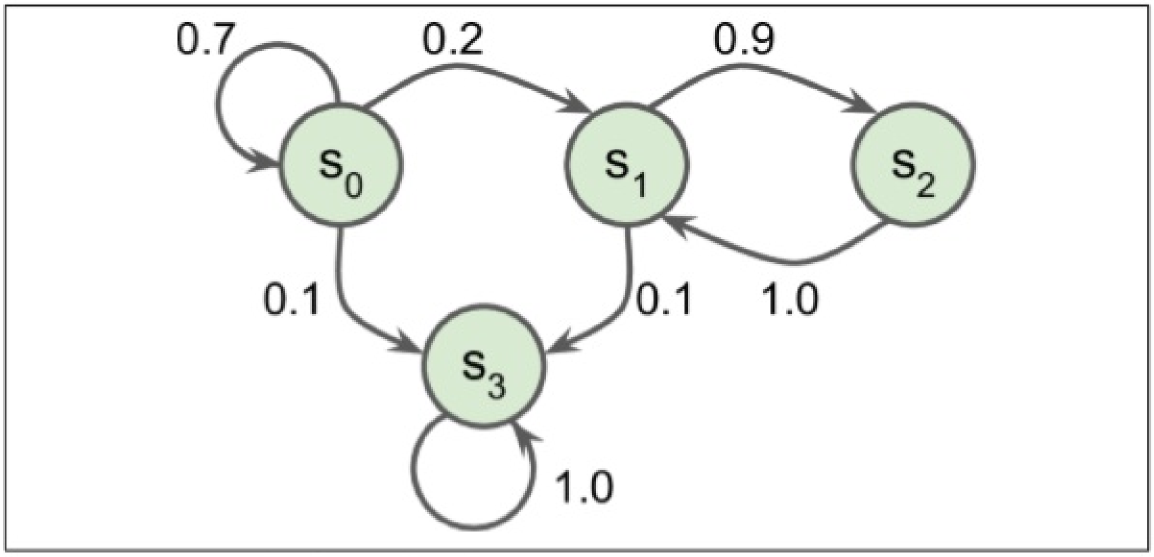
尽管它相对简单，但是该算法是非常强大的。你可以用它来解决更难的问题，而不仅仅是平衡一辆手推车上的平衡杆。事实上，AlgPaGo 是基于类似的 PG 算法（加上蒙特卡罗树搜索，这超出了本书的范围）。

现在我们来看看另一个流行的算法。与 PG 算法直接尝试优化策略以增加奖励相反，我们现在看的算法是间接的：智能点学习去估计每个状态的的每个行为未来衰减奖励的期望总和，或者在每个状态的的每个行为未来衰减奖励的期望和。然后，使用这些知识来决定如何行动。为了理解这些算法，我们必须首先介绍马尔可夫决策过程（MDP）。

马尔可夫决策过程

在二十世纪初，数学家 Andrey Markov 研究了没有记忆的随机过程，称为马尔可夫链。这样的过程具有固定数量的状态，并且在每个步骤中随机地从一个状态演化到另一个状态。它从状态 `s` 演变为状态 `s'` 的概率是固定的，它只依赖于 `(s, s')` 对，而不是依赖于过去的状态（系统没有记忆）。

图 16-7 展示了一个具有四个状态的马尔可夫链的例子。假设该过程从状态 `s0` 开始，并且在下一步骤中有 70% 的概率保持在该状态不变中。最终，它必然离开那个状态，并且永远不会回来，因为没有其他状态回到 `s0`。如果它进入状态 `s1`，那么它很可能会进入状态 `s2`（90% 的概率），然后立即回到状态 `s1`（以 100% 的概率）。它可以在这两个状态之间交替多次，但最终它会落入状态 `s3` 并永远留在那里（这是一个终端状态）。马尔可夫链可以有非常不同的应用，它们在热力学、化学、统计学等方面有着广泛的应用。



马尔可夫决策过程最初是在 20 世纪 50 年代由 Richard Bellman 描述的。它们类似于马尔可夫链，但有一个连接：在每一步中，一个智能点可以选择几种可能的动作中的一个，并且转移概率取决于所选择的动作。此外，一些状态转换返回一些奖励（正或负），智能点的目标是找到一个策略，随着时间的推移将最大限度地提高奖励。

例如，图 16-8 中所示的 MDP 在每个步骤中具有三个状态和三个可能的离散动作。如果从状态 `s0` 开始，随着时间的推移可以在动作 `A0`、`A1` 或 `A2` 之间进行选择。如果它选择动作 `A1`，它就保持在状态 `s0` 中，并且没有任何奖励。因此，如果愿意的话，它可以决定永远呆在那里。但是，如果它选择动作 `A0`，它有 70% 的概率获得 10 奖励，并保持在状态 `s0`。然后，它可以一次又一次地尝试获得尽可能多的奖励。但它将在状态 `s1` 中结束这样的行为。

在状态`S1`中，它只有两种可能的动作：`A0`或`A1`。它可以通过反复选择动作`A1`来选择停留，或者它可以选择移动到状态`S2`并得到 -50 奖励。在状态`S3`中，除了采取行动`A1`之外，别无选择，这将最有可能引导它回到状态`S0`，在途中获得 40 的奖励。通过观察这个 MDP，你能猜出哪一个策略会随着时间的推移而获得最大的回报吗？在状态`S0`中，清楚地知道`A0`是最好的选择，在状态`S3`中，智能点别无选择，只能采取行动`A1`，但是在状态`S1`中，智能点否应该保持不动（`A0`）或通过火（`A2`），这是不明确的。

Bellman 找到了一种估计任何状态`s`的最佳状态值的方法，他提出了`V(s)`，它是智能点在其达到最佳状态`s`后所有衰减的总和的平均期望。他表明，如果智能点的行为最佳，那么贝尔曼最优性公式适用（见公式 16-1）。这个递归公式表示，如果代理最优地运行，那么当前状态的最优值等于在采取一个最优动作之后平均得到的奖励，加上该动作可能导致的所有可能的下一个状态的期望最优值。

$$V^*(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma . V^*(s')]$$

其中：

- `T`为智能点选择动作`a`时从状态`s`到状态`s'`的概率
- `R`为智能点选择以动作`a`从状态`s`到状态`s'`的过程中得到的奖励
- `γ`为衰减率

这个等式直接引出了一种算法，该算法可以精确估计每个可能状态的最优状态值：首先将所有状态值估计初始化为零，然后用数值迭代算法迭代更新它们（见公式 16-2）。一个显著的结果是，给定足够的时间，这些估计保证收敛到最优状态值，对应于最优策略。

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma . V_k(s')]$$

其中：

- `V`是在`k`次算法迭代对状态`s`的估计

该算法是动态规划的一个例子，它将了一个复杂的问题（在这种情况下，估计潜在的未来衰减奖励的总和）变为可处理的子问题，可以迭代地处理（在这种情况下，找到最大化平均报酬与下一个衰减状态值的和的动作）

了解最佳状态值可能是有用的，特别是评估策略，但它没有明确地告诉智能点要做什么。幸运的是，Bellman 发现了一种非常类似的算法来估计最优状态动作值，通常称为 Q 值。状态行动`(S, A)`对的最优 Q 值，记为`Q(s, a)`，是智能点在选择动作 A 到达状态 S`之后平均期望的衰减的总和，但是是在它看到这个动作的结果之前，假设它在该动作之后起到最佳的作用。

下面是它的工作原理：再次，通过初始化所有的 Q 值估计为零，然后使用 Q 值迭代算法更新它们（参见公式 16-3）。

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma \cdot \max_{a'} Q_k(s',a')\right]$$

一旦你有了最佳的 Q 值，定义最优的策略 `π*(s)` ，它是微不足道的：当智能点处于状态 `s` 时，它应该选择具有最高 Q 值的动作，用于该状态。

让我们把这个算法应用到图 16-8 所示的 MDP 中。首先，我们需要定义 MDP：

```
nan=np.nan  # 代表不可能的动作
T = np.array([  # shape=[s, a, s']
    [[0.7, 0.3, 0.0], [1.0, 0.0, 0.0], [0.8, 0.2, 0.0]],
    [[0.0, 1.0, 0.0], [nan, nan, nan], [0.0, 0.0, 1.0]],
    [[nan, nan, nan], [0.8, 0.1, 0.1], [nan, nan, nan]],    ])
R = np.array([  # shape=[s, a, s']
    [[10., 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]],
    [[10., 0.0, 0.0], [nan, nan, nan], [0.0, 0.0, -50.]],
    [[nan, nan, nan], [40., 0.0, 0.0], [nan, nan, nan]],    ])
possible_actions = [[0, 1, 2], [0, 2], [1]]
```

让我们运行 Q 值迭代算法

```
Q = np.full((3, 3), -np.inf)  # -inf 对应着不可能的动作
for state, actions in enumerate(possible_actions):
    Q[state, actions] = 0.0  # 对所有可能的动作初始化为0.0
learning_rate = 0.01
discount_rate = 0.95
n_iterations = 100
for iteration in range(n_iterations):
    Q_prev = Q.copy()
    for s in range(3):
        for a in possible_actions[s]:
            Q[s, a] = np.sum([T[s, a, sp] * (R[s, a, sp] + discount_rate * np.max(Q_pr
ev[sp]))
            for sp in range(3)])
```

结果的 Q 值类似于如下：

```
>>> Q
array([[ 21.89498982,  20.80024033,  16.86353093],
        [  1.11669335,         -inf,   1.17573546],
        [        -inf,  53.86946068,         -inf]])
>>> np.argmax(Q, axis=1)  # 每一状态的最优动作
array([0, 2, 1])
```

这给我们这个 MDP 的最佳策略，当使用 0.95 的衰减率时：在状态 `s0` 选择动作 `A0` ，在状态 `s1` 选择动作 `A2` （通过火焰！）在状态 `s2` 中选择动作 `A1` （唯一可能的动作）。有趣的是，如果你把衰减率降低到 0.9，最优的策略改变：在状态 `s1` 中，最好的动作变成 `A0` （保持不变；不通过火）。这是有道理的，因为如果你认为现在比未来更重要，那么未来奖励的前景是不值得立刻经历痛苦的。

时间差异学习与 Q 学习

具有离散动作的强化学习问题通常可以被建模为马尔可夫决策过程，但是智能点最初不知道转移概率是什么（它不知道`T`），并且它不知道奖励会是什么（它不知道`R`）。它必须经历每一个状态和每一次转变并且至少知道一次奖励，并且如果要对转移概率进行合理的估计，就必须经历多次。

时间差分学习（TD 学习）算法与数值迭代算法非常类似，但考虑到智能点仅具有 MDP 的部分知识。一般来说，我们假设智能点最初只知道可能的状态和动作，没有更多了。智能点使用探索策略，例如，纯粹的随机策略来探索 MDP，并且随着它的发展，TD 学习算法基于实际观察到的转换和奖励来更新状态值的估计（见公式 16-4）。

$$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + \alpha\left(r + \gamma \, . \, V_k(s')\right)$$

其中：

`a`是学习率（例如 0.01）

TD 学习与随机梯度下降有许多相似之处，特别是它一次处理一个样本的行为。就像 SGD 一样，只有当你逐渐降低学习速率时，它才能真正收敛（否则它将在极值点震荡）。

对于每个状态`s`，该算法只跟踪智能点离开该状态时立即获得的奖励的平均值，再加上它期望稍后得到的奖励（假设它的行为最佳）。

类似地，Q 学习算法是 Q 值迭代算法的自适应版本，其适应过渡概率和回报在初始未知的情况（见公式16-5）。

$$Q_{k+1}(s,a) \leftarrow (1-\alpha)Q_k(s,a) + \alpha\left(r + \gamma \, . \, \max_{a'} \, Q_k(s',a')\right)$$

对于每一个状态动作对`(s,a)`，该算法跟踪智能点在以动作`A`离开状态`S`时获得的奖励的运行平均值`R`，加上它期望稍后得到的奖励。由于目标策略将最优地运行，所以我们取下一状态的 Q 值估计的最大值。

以下是如何实现 Q 学习：

```python
import numpy.random as rnd
learning_rate0 = 0.05
learning_rate_decay = 0.1
n_iterations = 20000
s = 0 # 在状态 0开始
Q = np.full((3, 3), -np.inf)  # -inf 对应着不可能的动作
for state, actions in enumerate(possible_actions):
    Q[state, actions] = 0.0  # 对于所有可能的动作初始化为 0.0
for iteration in range(n_iterations):
    a = rnd.choice(possible_actions[s])  # 随机选择动作
    sp = rnd.choice(range(3), p=T[s, a]) # 使用 T[s, a] 挑选下一状态
    reward = R[s, a, sp]
    learning_rate = learning_rate0 / (1 + iteration * learning_rate_decay)
    Q[s, a] = learning_rate * Q[s, a] + (1 - learning_rate) * (reward + discount_rate
* np.max(Q[sp]))
    s = sp # 移动至下一状态
```

给定足够的迭代，该算法将收敛到最优 Q 值。这被称为关闭策略算法，因为正在训练的策略不是正在执行的策略。令人惊讶的是，该算法能够通过观察智能点行为随机学习（例如学习当你的老师是一个醉猴子时打高尔夫球）最佳策略。我们能做得更好吗？

探索策略

当然，只有在探索策略充分探索 MDP 的情况下，Q 学习才能起作用。尽管一个纯粹的随机策略保证最终访问每一个状态和每个转换多次，但可能需要很长的时间这样做。因此，一个更好的选择是使用 ε 贪婪策略：在每个步骤中，它以概率 `ε` 随机地或以概率为 `1-ε` 贪婪地（选择具有最高 Q 值的动作）。ε 贪婪策略的优点（与完全随机策略相比）是，它将花费越来越多的时间来探索环境中有趣的部分，因为 Q 值估计越来越好，同时仍花费一些时间访问 MDP 的未知区域。以 `ε` 为很高的值（例如，1）开始，然后逐渐减小它（例如，下降到 0.05）是很常见的。

可选择的，相比于依赖于探索的机会，另一种方法是鼓励探索政策来尝试它以前没有尝试过的行动。这可以被实现为附加于 Q 值估计的奖金，如公式 16-6 所示。

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\left(r + \gamma \cdot \max_{\alpha'} f(Q(s',a'), N(s',a'))\right)$$

其中：

- `N` 计算了在状态 `s` 时选择动作 `a` 的次数
- `f` 是一个探索函数，例如 `f=q+K/(1+n)`，其中 `K` 是一个好奇超参数，它测量智能点被吸引到未知者的程度。

近似 Q 学习

Q 学习的主要问题是，它不能很好地扩展到具有许多状态和动作的大（甚至中等）的 MDP。试着用 Q 学习来训练一个智能点去玩 Ms. Pac-Man。Ms. Pac-Man 可以吃超过 250 粒粒子，每一粒都可以存在或不存在（即已经吃过）。因此，可能状态的数目大于 2 的 250 次幂，约等于 10 的 75 次幂（并且这是考虑颗粒的可能状态）。这比在可观测的宇宙中的原子要多得多，所以你绝对无法追踪每一个 Q 值的估计值。

解决方案是找到一个函数，使用可管理数量的参数来近似 Q 值。这被称为近似 Q 学习。多年来，人们都是手工在状态中提取并线性组合特征（例如，最近的鬼的距离，它们的方向等）来估计 Q 值，但是 DeepMind 表明使用深度神经网络可以工作得更好，特别是对于复杂的问题。它不需要任何特征工程。用于估计 Q 值的 DNN 被称为深度 Q 网络（DQN），并且使用近似 Q 学习的 DQN 被称为深度 Q 学习。

在本章的剩余部分，我们将使用深度 Q 学习来训练一个智能点去玩 Ms. Pac-Man，就像 DeepMind 在 2013 所做的那样。代码可以很容易地调整，调整后学习去玩大多数 Atari 游戏的效果都相当好。在大多数动作游戏中，它可以达到超人的技能，但它在长卷游戏中却不太好。

学习去使用深度 Q 学习来玩 Ms.Pac-Man

由于我们将使用 Atari 环境，我们必须首先安装 OpenAI gym 的 Atari 环境依赖项。当需要玩其他的时候，我们也会为你想玩的其他 OpenAI gym 环境安装依赖项。在 macOS 上，假设你已经安装了 Homebrew 程序，你需要运行：

```
$ brew install cmake boost boost-python sdl2 swig wget
```

在 Ubuntu 上，输入以下命令（如果使用 Python 2，用 Python 替换 Python 3）：

```
$ apt-get install -y python3-numpy python3-dev cmake zlib1g-dev libjpeg-dev\    xvfb l
ibav-tools xorg-dev python3-opengl libboost-all-dev libsdl2-dev swig
```

随后安装额外的 python 包：

```
$ pip3 install --upgrade 'gym[all]'
```

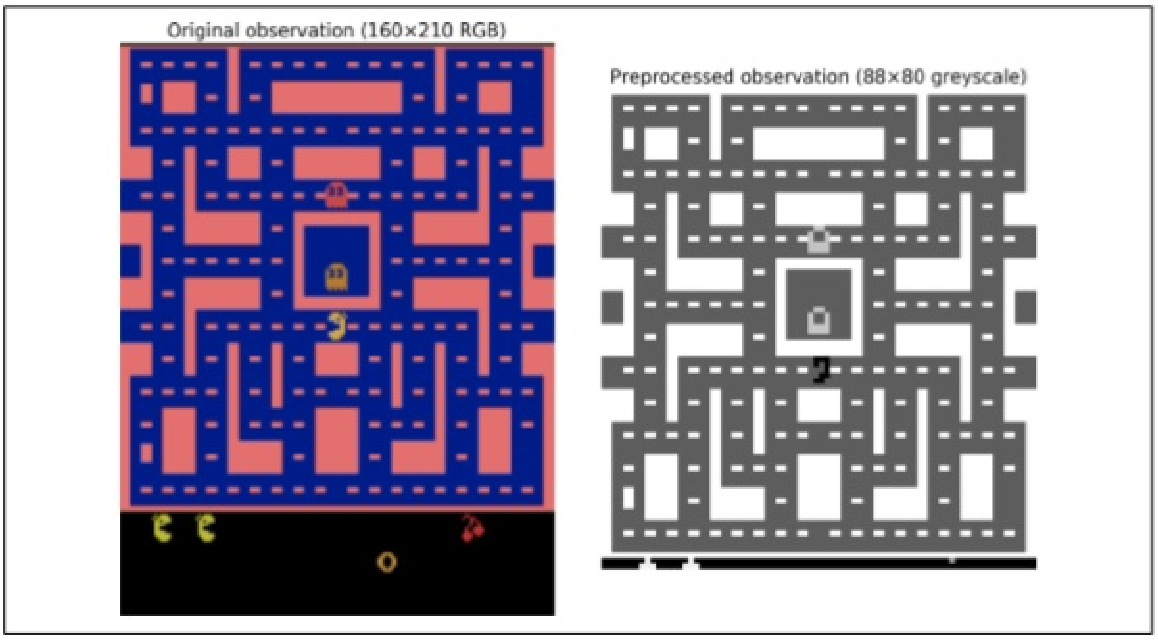
如果一切顺利，你应该能够创造一个 Ms.Pac-Man 环境：

```
>>> env = gym.make("MsPacman-v0")
>>> obs = env.reset()
>>> obs.shape  # [长，宽，通道]
(210, 160, 3)
>>> env.action_space
Discrete(9)
```

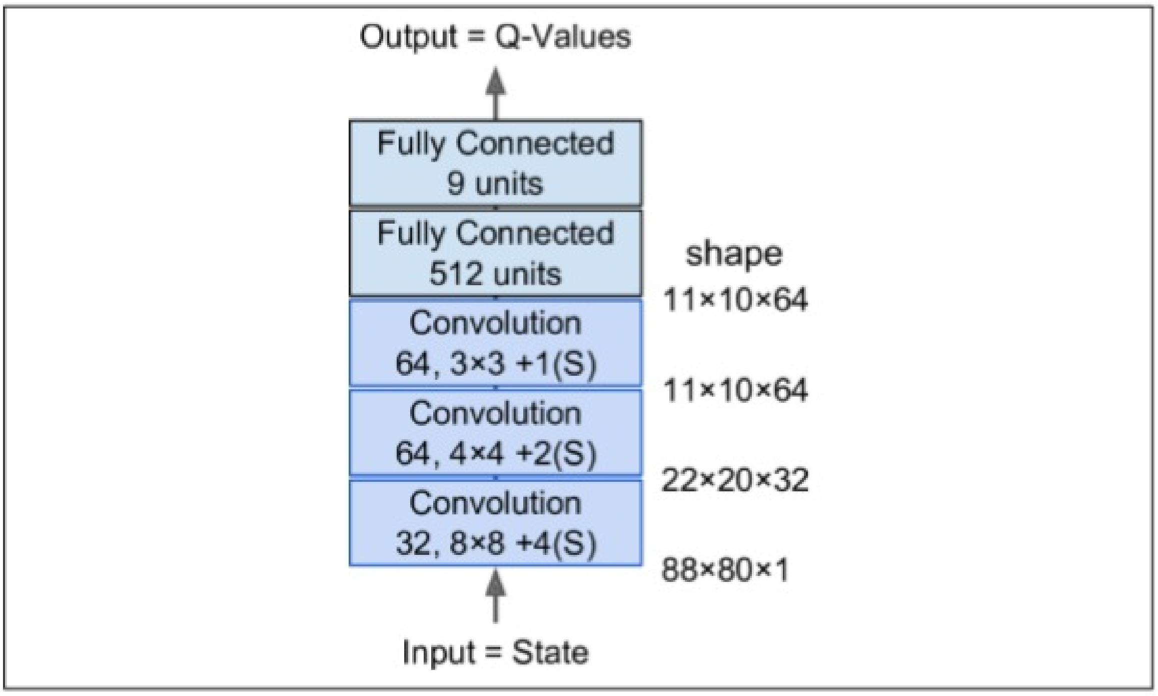
正如你所看到的，有九个离散动作可用，它对应于操纵杆的九个可能位置（左、右、上、下、中、左上等），观察结果是 Atari 屏幕的截图（见图 16-9，左），表示为 3D Numpy 矩阵。这些图像有点大，所以我们将创建一个小的预处理函数，将图像裁剪并缩小到 `88×80` 像素，将其转换成灰度，并提高 Ms.Pac-Man 的对比度。这将减少 DQN 所需的计算量，并加快培训练。

```
def preprocess_observation(obs):
    img = obs[1:176:2, ::2] # 裁剪
    img = img.mean(axis=2) # 灰度化
    img[img==mspacman_color] = 0 # 提升对比度
    img = (img - 128) / 128 - 1 # 正则化为-1到1.
    return img.reshape(88, 80, 1)
```

过程的结果如图 16-9 所示（右）。



接下来，让我们创建 DQN。它可以只取一个状态动作对 `(s,a)` 作为输入，并输出相应的 Q 值 `Q(s,a)` 的估计值，但是由于动作是离散的，所以使用只使用状态 `s` 作为输入并输出每个动作的一个 Q 值估计的神经网络是更方便的。DQN 将由三个卷积层组成，接着是两个全连接层，其中包括输出层（如图 16-10）。



正如我们将看到的，我们将使用的训练算法需要两个具有相同架构（但不同参数）的 DQN：一个将在训练期间用于驱动 Ms.Pac-Man（演员），另一个将观看演员并从其试验和错误中学习（评论家）。每隔一定时间，我们就把评论家的反馈给演员看。因为我们需要两个相同的 DQN，所以我们将创建一个 `q_network()` 函数来构建它们：

```
from tensorflow.contrib.layers import convolution2d, fully_connected
input_height = 88
input_width = 80
input_channels = 1
conv_n_maps = [32, 64, 64]
conv_kernel_sizes = [(8,8), (4,4), (3,3)]
conv_strides = [4, 2, 1]
conv_paddings = ["SAME"]*3
conv_activation = [tf.nn.relu]*3
n_hidden_in = 64 * 11 * 10  # conv3 有 64 个 11x10 映射
each n_hidden = 512
hidden_activation = tf.nn.relu n_outputs = env.action_space.n  # 9个离散动作
initializer = tf.contrib.layers.variance_scaling_initializer()

def q_network(X_state, scope):
    prev_layer = X_state
    conv_layers = []
    with tf.variable_scope(scope) as scope:
        for n_maps, kernel_size, stride, padding, activation in zip(conv_n_maps, conv_
kernel_sizes,                 conv_strides,conv_paddings, conv_activation):

            prev_layer = convolution2d(prev_layer, num_outputs=n_maps, kernel_size=ker
nel_size,stride=stride, padding=padding, activation_fn=activation,weights_initializer=
initializer)          conv_layers.append(prev_layer)
        last_conv_layer_flat = tf.reshape(prev_layer, shape=[-1, n_hidden_in])
        hidden = fully_connected(last_conv_layer_flat, n_hidden,activation_fn=hidden_a
ctivation,                                                                        weig
hts_initializer=initializer)                outputs = fully_connected(hidden, n_output
s, activation_fn=None,weights_initializer=initializer)
    trainable_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,scope=scope.na
me)
    trainable_vars_by_name = {var.name[len(scope.name):]: var for var in trainable_var
s}
    return outputs, trainable_vars_by_name
```

该代码的第一部分定义了DQN体系结构的超参数。然后 `q_network()` 函数创建 DQN，将环境的状态 `X_state` 作为输入，以及变量范围的名称。请注意，我们将只使用一个观察来表示环境的状态，因为几乎没有隐藏的状态（除了闪烁的物体和鬼魂的方向）。

`trainable_vars_by_name` 字典收集了所有 DQN 的可训练变量。当我们创建操作以将评论家 DQN 复制到演员 DQN 时，这将是有用的。字典的键是变量的名称，去掉与范围名称相对应的前缀的一部分。看起来像这样：

```
>>> trainable_vars_by_name
{'/Conv/biases:0': <tensorflow.python.ops.variables.Variable at 0x121cf7b50>, '/Conv/w
eights:0': <tensorflow.python.ops.variables.Variable...>,
'/Conv_1/biases:0': <tensorflow.python.ops.variables.Variable...>, '/Conv_1/weights:0'
: <tensorflow.python.ops.variables.Variable...>, '/Conv_2/biases:0': <tensorflow.pytho
n.ops.variables.Variable...>, '/Conv_2/weights:0': <tensorflow.python.ops.variables.Va
riable...>, '/fully_connected/biases:0': <tensorflow.python.ops.variables.Variable...>
, '/fully_connected/weights:0': <tensorflow.python.ops.variables.Variable...>, '/fully
_connected_1/biases:0': <tensorflow.python.ops.variables.Variable...>, '/fully_connect
ed_1/weights:0': <tensorflow.python.ops.variables.Variable...>}
```

现在让我们为两个 DQN 创建输入占位符，以及复制评论家 DQN 给演员 DQN 的操作：

```
X_state = tf.placeholder(tf.float32, shape=[None, input_height, input_width,input_chan
nels])                    actor_q_values, actor_vars = q_network(X_state, scope="q_net
works/actor")
critic_q_values, critic_vars = q_network(X_state, scope="q_networks/critic")
copy_ops = [actor_var.assign(critic_vars[var_name])  for var_name, actor_var in actor_
vars.items()]
copy_critic_to_actor = tf.group(*copy_ops)
```

让我们后退一步：我们现在有两个 DQN，它们都能够将环境状态（即预处理观察）作为输入，并输出在该状态下的每一个可能的动作的估计 Q 值。另外，我们有一个名为 `copy_critic_to_actor` 的操作，将评论家 DQN 的所有可训练变量复制到演员 DQN。我们使用 TensorFlow 的 `tf.group()` 函数将所有赋值操作分组到一个方便的操作中。

演员 DQN 可以用来扮演 Ms.Pac-Man（最初非常糟糕）。正如前面所讨论的，你希望它足够深入地探究游戏，所以通常情况下你想将它用 ε 贪婪策略或另一种探索策略相结合。

但是评论家 DQN 呢？它如何去学习玩游戏？简而言之，它将试图使其预测的 Q 值去匹配演员通过其经验的游戏估计的 Q 值。具体来说，我们将让演员玩一段时间，把所有的经验保存在回放记忆中。每个内存将是一个 5 元组（状态、动作、下一状态、奖励、继续），其中“继续”项在游戏结束时等于 0，否则为 1。接下来，我们定期地从回放存储器中采样一批记忆，并且我们将估计这些存储器中的 Q 值。最后，我们将使用监督学习技术训练评论家 DQN 去预测这些 Q 值。每隔几个训练周期，我们会把评论家 DQN 复制到演员 DQN。就这样！公式 16-7 示出了用于训练评论家 DQN 的损失函数：

$$J(\theta_{\text{critic}}) = \frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} - Q\left(s^{(i)}, a^{(i)}, \theta_{\text{critic}}\right)\right)^2$$

$$\text{with } y^{(i)} = r^{(i)} + \gamma \cdot \max_{a'} Q\left(s'^{(i)}, a', \theta_{\text{actor}}\right)$$

其中：

- `s` ，`A` ，`R` 和 `S'` 分别为状态，行为，回报，和在下一状态在存储器中采样来的第 `i` 个记忆
- `m` 是记忆批处理的长度
- `Oc` 和 `Oa` 为评论家和演员的参数
- `Q` 是评论家 DQN 对第 `i` 记忆状态行为 Q 值的预测
- `Q'` 是演员 DQN 在选择动作 `A` 时的下一状态 `S'` 的期望 Q 值的预测
- `y` 是第 `i` 记忆的目标 Q 值，注意，它等同于演员实际观察到的奖励，再加上演员对如果它能发挥最佳效果（据它所知），未来的回报应该是什么的预测。
- `J` 为训练评论家 DQN 的损失函数。正如你所看到的，这只是由演员 DQN 估计的目标 Q 值 `y` 和评论家 DQN 对这些 Q 值的预测之间的均方误差。

回放记忆是可选的，但强烈推荐使它存在。没有它，你会训练评论家 DQN 使用连续的经验，这可能是相关的。这将引入大量的偏差并且减慢训练算法的收敛性。通过使用回放记忆，我们确保馈送到训练算法的存储器可以是不相关的。

让我们添加评论家 DQN 的训练操作。首先，我们需要能够计算其在存储器批处理中的每个状态动作的预测 Q 值。由于 DQN 为每一个可能的动作输出一个 Q 值，所以我们只需要保持与在该存储器中实际选择的动作相对应的 Q 值。为此，我们将把动作转换成一个热向量（记住这是一个满是 0 的向量，除了第 `i` 个索引中的1），并乘以 Q 值：这将删除所有与记忆动作对应的 Q 值外的 Q 值。然后只对第一轴求和，以获得每个存储器所需的 Q 值预测。

```
X_action = tf.placeholder(tf.int32, shape=[None])
q_value = tf.reduce_sum(critic_q_values * tf.one_hot(X_action, n_outputs), axis=1, kee
p_dims=True)
```

接下来，让我们添加训练操作，假设目标Q值将通过占位符馈入。我们还创建了一个不可训练的变量 `global_step` 。优化器的 `minimize()` 操作将负责增加它。另外，我们创建了 `init` 操作和 `Saver` 。

```
y = tf.placeholder(tf.float32, shape=[None, 1])
cost = tf.reduce_mean(tf.square(y - q_value))
global_step = tf.Variable(0, trainable=False, name='global_step')
optimizer = tf.train.AdamOptimizer(learning_rate)
training_op = optimizer.minimize(cost, global_step=global_step)
init = tf.global_variables_initializer() saver = tf.train.Saver()
```

这就是训练阶段的情况。在我们查看执行阶段之前，我们需要一些工具。首先，让我们从回放记忆开始。我们将使用一个 `deque` 列表，因为在将数据推送到队列中并在达到最大内存大小时从列表的末尾弹出它们使是非常有效的。我们还将编写一个小函数来随机地从回放记忆中采样一批处理：

```
from collections import deque

replay_memory_size = 10000
replay_memory = deque([], maxlen=replay_memory_size)

def sample_memories(batch_size):
    indices = rnd.permutation(len(replay_memory))[:batch_size]
    cols = [[], [], [], [], []] # state, action, reward, next_state, continue
    for idx in indices:
        memory = replay_memory[idx]
        for col, value in zip(cols, memory):
            col.append(value)
    cols = [np.array(col) for col in cols]
    return (cols[0], cols[1], cols[2].reshape(-1, 1), cols[3],cols[4].reshape(-1, 1))
```

接下来，我们需要演员来探索游戏。我们使用 ε 贪婪策略，并在 50000 个训练步骤中逐步将 `ε` 从 1 降低到 0.05。

```
eps_min = 0.05
eps_max = 1.0
eps_decay_steps = 50000
def epsilon_greedy(q_values, step):
    epsilon = max(eps_min, eps_max - (eps_max-eps_min) * step/eps_decay_steps)
     if rnd.rand() < epsilon:
        return rnd.randint(n_outputs) # 随机动作
    else:
        return np.argmax(q_values) # 最优动作
```

就是这样！我们准备好开始训练了。执行阶段不包含太复杂的东西，但它有点长，所以深呼吸。准备好了吗？来次够！首先，让我们初始化几个变量：

```
n_steps = 100000  # 总的训练步长
training_start = 1000  # 在游戏1000次迭代后开始训练
training_interval = 3  # 每3次迭代训练一次
save_steps = 50  # 每50训练步长保存模型
copy_steps = 25  # 每25训练步长后复制评论家Q值到演员
discount_rate = 0.95
skip_start = 90  # 跳过游戏开始(只是等待时间)
batch_size = 50
iteration = 0  # 游戏迭代
checkpoint_path = "./my_dqn.ckpt"
done = True # env 需要被重置
```

接下来，让我们打开会话并开始训练：

```
with tf.Session() as sess:
    if os.path.isfile(checkpoint_path):
        saver.restore(sess, checkpoint_path)
    else:
        init.run()
    while True:
        step = global_step.eval()
        if step >= n_steps:
            break
        iteration += 1
        if done: # 游戏结束，重来
            obs = env.reset()
            for skip in range(skip_start): # 跳过游戏开头
                obs, reward, done, info = env.step(0)
                state = preprocess_observation(obs)

        # 演员评估要干什么
        q_values = actor_q_values.eval(feed_dict={X_state: [state]})
        action = epsilon_greedy(q_values, step)

        # 演员开始玩游戏
        obs, reward, done, info = env.step(action)
        next_state = preprocess_observation(obs)

        # 让我们记下来刚才发生了啥
        replay_memory.append((state, action, reward, next_state, 1.0 - done))        s
tate = next_state

        if iteration < training_start or iteration % training_interval != 0:
      continue

        # 评论家学习
        X_state_val, X_action_val, rewards, X_next_state_val, continues = (
 sample_memories(batch_size))
        next_q_values = actor_q_values.eval( feed_dict={X_state: X_next_state_val})

        max_next_q_values = np.max(next_q_values, axis=1, keepdims=True)
        y_val = rewards + continues * discount_rate * max_next_q_values
        training_op.run(feed_dict={X_state: X_state_val,X_action: X_action_val, y: y_v
al})

        # 复制评论家Q值到演员
        if step % copy_steps == 0:
            copy_critic_to_actor.run()

        # 保存模型
        if step % save_steps == 0:
            saver.save(sess, checkpoint_path)
```

如果检查点文件存在，我们就开始恢复模型，否则我们只需初始化变量。然后，主循环开始，其中 `iteration` 计算从程序开始以来游戏步骤的总数，同时 `step` 计算从训练开始的训练步骤的总数（如果恢复了检查点，也恢复全局步骤）。然后代码重置游戏（跳过第一个无聊的等待游戏的步骤，这步骤啥都没有）。接下来，演员评估该做什么，并且玩游戏，并且它的经验被存储在回放记忆中。然后，每隔一段时间（热身期后），评论家开始一个训练步骤。它采样一批回放记忆，并要求演员估计下一状态的所有动作的Q值，并应用公式 16-7 来计算目标 Q 值 `y_val`.这里唯一棘手的部分是，我们必须将下一个状态的 Q 值乘以 `continues` 向量，以将对应于游戏结束的记忆 Q 值清零。接下来，我们进行训练操作，以提高评论家预测 Q 值的能力。最后，我们定期将评论家的 Q 值复制给演员，然后保存模型。

不幸的是，训练过程是非常缓慢的：如果你使用你的破笔记本电脑进行训练的话，想让 Ms. Pac-Man 变好一点点你得花好几天，如果你看看学习曲线，计算一下每次的平均奖励，你会发现到它是非常嘈杂的。在某些情况下，很长一段时间内可能没有明显的进展，直到智能点学会在合理的时间内生存。如前所述，一种解决方案是将尽可能多的先验知识注入到模型中（例如，通过预处理、奖励等），也可以尝试通过首先训练它来模仿基本策略来引导模型。在任何情况下，RL仍然需要相当多的耐心和调整，但最终结果是非常令人兴奋的。

练习

1. 你怎样去定义强化学习？它与传统的监督以及非监督学习有什么不同？
2. 你能想到什么本章没有提到过的强化学习应用？智能体是什么？什么是可能的动作，什么是奖励？
3. 什么是衰减率？如果你修改了衰减率那最优策略会变化吗？
4. 你怎么去定义强化学习智能体的表现？
5. 什么是信用评估问题？它怎么出现的？你怎么解决？
6. 使用回放记忆的目的是什么？
7. 什么是闭策略 RL 算法？
8. 使用深度 Q 学习来处理 OpenAI gym 的“BypedalWalker-v2” 。QNET 不需要对这个任务使用非常深的网络。
9. 使用策略梯度训练智能体扮演 Pong，一个著名的 Atari 游戏（PANV0 在 OpenAI gym 的 Pong-v0）。注意：个人的观察不足以说明球的方向和速度。一种解决方案是一次将两次观测传递给神经网络策略。为了减少维度和加速训练，你必须预先处理这些图像（裁剪，调整大小，并将它们转换成黑白），并可能将它们合并成单个图像（例如去叠加它们）。
10. 如果你有大约 100 美元备用，你可以购买 Raspberry Pi 3 再加上一些便宜的机器人组件，在 PI 上安装 TensorFlow，然后让我们嗨起来~！举个例子，看看 Lukas Biewald 的这个有趣的帖子，或者看看 GoPiGo 或 BrickPi。为什么不尝试通过使用策略梯度训练机器人来构建真实的 cartpole ？或者造一个机器人蜘蛛，让它学会走路；当它接近某个目标时，给予奖励（你需要传感器来测量目标的距离）。唯一的限制就是你的想象力。

练习答案均在附录 A。

感谢

在我们结束这本书的最后一章之前，我想感谢你们读到最后一段。我真心希望你能像我写这本书一样愉快地阅读这本书，这对你的项目，或多或少都是有用的。

如果发现错误，请发送反馈。更一般地说，我很想知道你的想法，所以请不要犹豫，通过 O'Reilly 来与我联系，或者通过 ageron/handson-ml GITHUB 项目来练习。

对你来说，我最好的建议是练习和练习：如果你还没有做过这些练习，试着使用 Juyter notebook 参加所有的练习，加入 kaggle 网站或其他 ML 社区，看 ML 课程，阅读论文，参加会议，会见专家。您可能还想研究我们在本书中没有涉及的一些主题，包括推荐系统、聚类算法、异常检测算法和遗传算法。

我最大的希望是，这本书将激励你建立一个美妙的 ML 应用程序，这将有利于我们所有人！那会是什么呢？

> 2016 年 11 月 26 日，奥列伦·格伦
>
> 你的支持，是我们每个开源工作者的骄傲～